International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004

Scalable Multi-Application CI/CD Pipeline Management using Master Pipeline Architecture

Bhanuprakash Suravarapu

DevOps Engineer, Amazon Web Services Inc., USA

bhanuprakash.suravarapu6@gmail.com

Praneeth Ganta

Senior Software Engineer, LinkedIn, USA

praneethganta@gmail.com

Tulasi priya Vattikuti

Cloud Data Engineer, Medtronic, USA

tulcpriya@gmail.com

Vasu Babu Narra

Senior Release Manager, Uipath Inc., USA

narravasu77@gmail.com

ABSTRACT: Modern organizations often host tens of development teams, each requiring its own continuous integration and continuous delivery (CI/CD) pipeline. Manual creation and maintenance of these pipelines lead to platform team bottlenecks, inconsistent security practices, and limited governance at scale. This paper presents a "pipelines of pipelines" architecture that uses a master pipeline to automatically discover Terraform based applications in a shared repository and provision standardized CI/CD infrastructure for each team. The system supports multiple source providers (GitHub, GitLab, AWS CodeCommit, and Amazon S3), integrates security and compliance scanning (Terraform Validate, TFLint, Checkov, and TFSec), and enforces centralized approval workflows and audit trails. The design shows how centrally governed, self-service CI/CD can be achieved through automated discovery and templated provisioning rather than per-team manual configuration.

KEYWORDS: CI/CD, DevOps, Terraform, AWS CodePipeline, platform engineering, GitOps, automation.

I. INTRODUCTION

Large organizations increasingly adopt DevOps and infrastructure as code practices, yet their CI/CD implementations often scale poorly across many independent teams. Each team demands a tailored pipeline, while the platform group must manually configure source integration, build stages, infrastructure deployment, and security controls for every new project. This ticket-driven model does not scale linearly with organizational growth and leads to long lead times for new applications, inconsistent security posture, and high operational overhead.

At the same time, platform engineering and GitOps principles advocate self-service infrastructure and declarative change management. However, many organizations struggle to operationalize these ideas for dozens of application pipelines across heterogeneous source providers and compliance requirements. A key challenge is balancing team autonomy with centralized governance, security, and cost control.

This work addresses the problem of scalable multi-application CI/CD management in a single cloud environment. The goal is to provide development teams with self-service pipelines while enabling the platform team to enforce consistent patterns, security checks, and approval workflows. The central question is how pipeline provisioning and governance can be automated so that adding a new application imposes minimal marginal operational cost.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004

The contributions of this paper are threefold: (1) a master-pipeline architecture that automatically discovers Terraform-based applications and provisions standardized pipelines per application; (2) an implementation that supports multiple source providers, multi-tool validation, detailed planning and impact analysis, and controlled deployment with rollback; and (3) a discussion of design trade-offs, limitations, and directions for further enhancement of the pattern.

II. BACKGROUND AND RELATED WORK

Traditional CI/CD systems, such as Jenkins, provide flexible pipeline definitions but require significant operational effort to maintain build agents, plugins, and scaling policies. Cloud-hosted alternatives, including GitLab CI and GitHub Actions, simplify pipeline execution but typically treat each project in isolation, leaving cross-team governance and standardization largely manual. Cloud-native services such as AWS CodePipeline, Azure DevOps, and Google Cloud Build offer managed orchestration yet still require per-project configuration of stages, roles, and integrations.

Platform engineering practices advocate building internal platforms that offer paved-road workflows, where teams consume standardized infrastructure via self-service interfaces rather than bespoke pipelines. GitOps extends this model by treating the desired state as a version-controlled configuration, applied by automated controllers. Despite these advances, many organizations lack a practical mechanism to automatically instantiate and manage dozens of pipelines in a consistent way; the “last mile” from shared patterns to per-team pipelines often remains manual.

The approach presented in this paper builds on these ideas by introducing a master pipeline that treats per-team pipelines themselves as provisioned infrastructure. Applications are discovered declaratively via their Terraform configuration, and the required CI/CD infrastructure is synthesized from a single template, enabling centralized evolution of pipeline behavior with minimal per-team customization.

III. PROBLEM STATEMENT

A. Multi-Team CI/CD at Scale

In a large organization, each development team typically maintains its own application stack and expects an associated CI/CD pipeline to automate build, test, and deployment tasks. When pipelines are created manually, the platform team becomes a critical path for onboarding new projects: each pipeline requires configuring source providers, build projects, artifact storage, IAM roles, and security scanners. This consumes substantial platform-engineering capacity and delays application delivery.

B. Inconsistent Security and Governance

When teams define pipelines independently, security and compliance practices vary widely. Some pipelines include infrastructure-as-code scanning or approval workflows, while others do not. Platform teams cannot easily enforce consistent validation rules, cost controls, or audit logging across heterogeneous pipelines. Over time, this leads to configuration drift, inconsistent risk posture, and increased audit burden.

C. Requirements

To address these issues, a scalable solution must satisfy the following requirements:

- Provide self-service pipeline provisioning for multiple teams with minimal central intervention.
- Standardize security and quality checks across all pipelines, including infrastructure validation and policy enforcement.
- Support multiple Git providers to accommodate existing organizational choices.
- Maintain clear security boundaries, least-privilege access, and robust state management.
- Enable the platform team to evolve pipeline behavior centrally without re-implementing per-team configurations.

IV. SYSTEM ARCHITECTURE

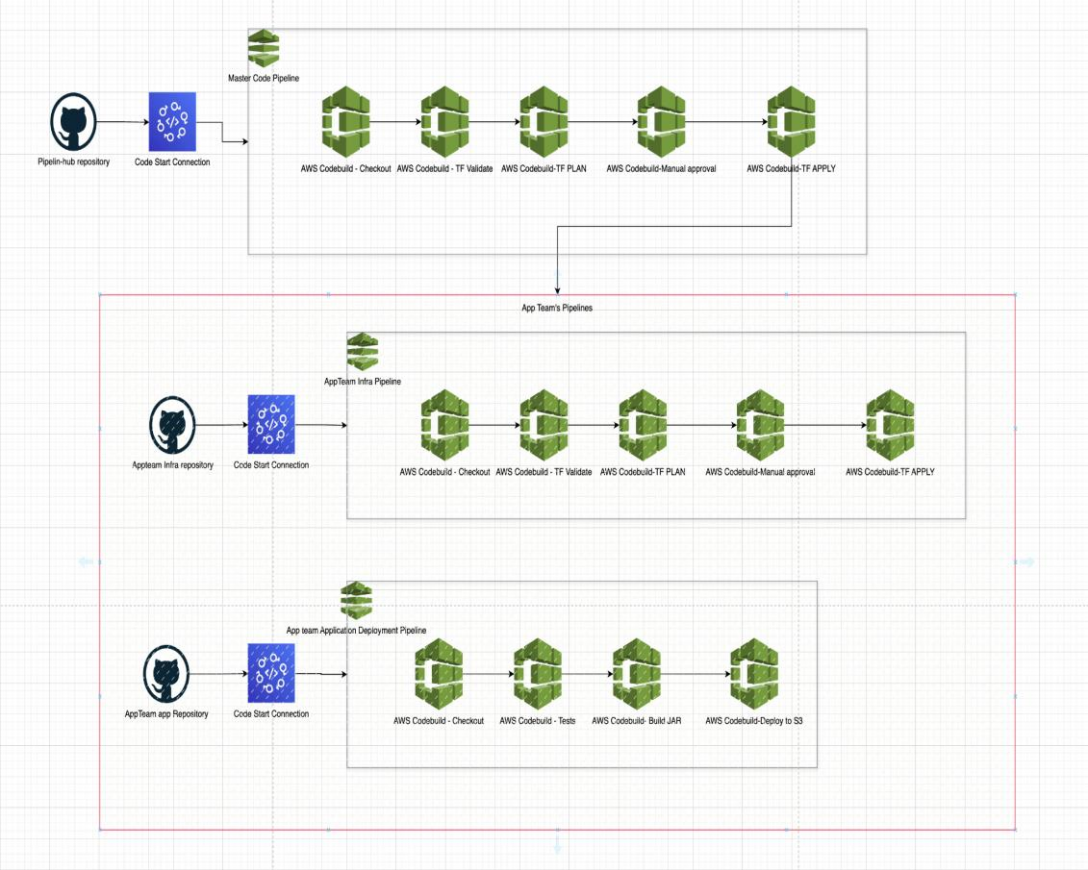
A. Overview

The proposed system implements a “master pipeline” that monitors a shared Git repository (the pipeline hub) and automatically provisions per-application CI/CD pipelines whenever Terraform configuration is detected. The master pipeline encapsulates source integration, discovery, validation, planning, governance, and deployment stages. Each discovered application receives its own infrastructure pipeline, parameterized by team-specific variables but derived from a common template.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004



B. Components

The architecture comprises five main components:

- Source Integration Layer: Connects to multiple Git providers (GitHub, GitLab, AWS CodeCommit, and Amazon S3) through cloud-native integrations and webhooks.
- Discovery Engine: Scans the repository tree for directories containing Terraform configuration files and treats each directory as a candidate application.
- Provisioning Engine: Generates and applies infrastructure templates, creating per-application pipelines, build projects, IAM roles, and state backends.
- Security and Validation Layer: Executes Terraform Validate, TFLint, Checkov, and TFSec against each application before provisioning changes.
- Governance Layer: Implements approval workflows, notifications, and audit trails, including cost and security impact analysis.

C. Data and Control Flow

The end-to-end workflow proceeds as follows:

1. A team commits the Terraform configuration for a new or existing application to the shared repository.
2. Source-provider webhooks or events trigger execution of the master pipeline.
3. The discovery engine scans the repository, identifies directories containing Terraform files, and selects applications for processing.
4. The validation layer runs syntax checks, linting, and security scans for each application.
5. For validated applications, the planning stage generates infrastructure plans and impact reports.
6. Governance workflows route plans for approval, with notifications sent to the platform, security, and application stakeholders.
7. Upon approval, the deployment stage applies Terraform plans to create or update the per-application pipelines and associated infrastructure.
8. Metadata about changes, approvals, and outcomes is persisted for audit and reporting.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004

V. IMPLEMENTATION

This section describes the concrete implementation of the master pipeline using shell scripts and Terraform. Only representative excerpts are shown; full scripts can be provided in an appendix.

A. Multi-Provider Source Integration

Source integration is implemented using conditional configuration for each supported provider. A single pipeline definition dynamically instantiates the appropriate source action based on configuration parameters, allowing the same master pipeline to ingest changes from GitHub, GitLab, CodeCommit, or S3. Provider-specific authentication is handled via cloud-native connections and IAM roles, and all source artifacts are encrypted in transit and at rest.

```
# GitHub Integration via CodeStar Connections
dynamic "action" {
 for_each = var.source_provider == "GitHub" ? [1] : []
 content {
  name             = "Source"
  category         = "Source"
  owner            = "AWS"
  provider         = "CodeStarSourceConnection"
  version          = "1"
  output_artifacts = ["SOURCE_ARTIFACT"]
  configuration = {
   ConnectionArn        = var.github_connection_arn
   FullRepositoryId     = var.github_repo_full_name
   BranchName           = var.pipeline_hub_branch
   OutputArtifactFormat = "CODE_ZIP"
  }
 }
}
# AWS CodeCommit Native Integration
dynamic "action" {
 for_each = var.source_provider == "CodeCommit" ? [1] : []
 content {
  name             = "Source"
  category         = "Source"
  owner            = "AWS"
  provider         = "CodeCommit"
  version          = "1"
  output_artifacts = ["SOURCE_ARTIFACT"]
  configuration = {
   RepositoryName       = aws_codecommit_repository.pipeline_hub[0].repository_name
   BranchName           = var.pipeline_hub_branch
   PollForSourceChanges = false  # EventBridge-driven
   OutputArtifactFormat = "CODE_ZIP"
  }
 }
}
```

Equivalent blocks exist for GitLab and S3, so the same master pipeline can be reused regardless of where the central repository is hosted.

B. Discovery and Validation

Applications are discovered by scanning the shared repository for directories containing Terraform configuration. Each directory with at least one .tf or .tf.json file is treated as a candidate application:

```
# buildspec.yaml - Automatic application discovery

for dir in */ ; do
```

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004

```
 stack=$(basename "$dir")
  # Check if directory contains Terraform files
 tf_files=$(find "$stack" -maxdepth 1 -name "*.tf" -o -name "*.tf.json" 2>/dev/null | wc -l)
 if [ "$tf_files" -eq 0 ]; then
   echo "Skipping $stack - no Terraform files found"
   continue
 fi
  echo "Processing application: $stack"
 # Run terraform validate/plan/apply for each application
done
```

This logic ensures that no out-of-band configuration is required: if a team adds Terraform to a folder, the system automatically considers it an application and moves it through the validation pipeline.
The validate_application function orchestrates syntax, quality, and security checks:

```
validate_application() {
 local app_name=$1
  echo "==== Terraform Syntax Validation for $app_name ===="
 terraform init -backend=false || { echo "Terraform init failed"; exit 1; }
 terraform validate || { echo "Terraform validate failed"; exit 1; }
  echo "==== TFLint Code Quality Analysis ===="
 tflint --init || { echo "TFLint init failed"; exit 1; }
 tflint --format=compact || { echo "TFLint analysis failed"; exit 1; }
  echo "==== Checkov Security Compliance Scanning ===="
 checkov --framework terraform -d . \
   --output cli \
   --soft-fail \
   --compact || echo "Checkov found security issues (non-blocking)"
  echo "==== TFSec Terraform Security Analysis ===="
 if command -v tfsec >/dev/null 2>&1; then
   tfsec . --format lovely --soft-fail || echo "TFSec found security issues (non-blocking)"
 else
   echo "TFSec not available - installing..."
   curl -s https://raw.githubusercontent.com/aquasecurity/tfsec/master/scripts/install_linux.sh | bash
   tfsec . --format lovely --soft-fail || echo "TFSec found security issues (non-blocking)"
 fi
}
```

Terraform Validate and TFLint act as hard gates, while Checkov and TFSec are typically run in soft-fail mode, surfacing issues without automatically blocking progress unless organizational policy requires it.

C. Planning and Impact Analysis

For each application that passes validation, the system generates a Terraform plan and analyzes its impact before deployment:

```
plan_infrastructure() {
 local app_name=$1
  echo "==== Terraform Planning for $app_name ===="
  # Install git-remote-codecommit for state backend access
 pip3 install git-remote-codecommit || { echo "Failed to install git-remote-codecommit"; exit 1; }
  # Initialize with remote state backend
 terraform init || { echo "Terraform init failed for $app_name"; exit 1; }
  # Generate detailed execution plan
 terraform plan \
   -out=tfplan \
   -input=false \
   -no-color \
   -detailed-exitcode || handle_plan_result $? "$app_name"
```

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004

```
 # Generate machine-readable plan output
terraform show -json tfplan > tfplan.json || { echo "Failed to generate JSON plan"; exit 1; }
 # Generate human-readable plan summary
terraform show tfplan > tfplan.txt || { echo "Failed to generate text plan"; exit 1; }
 # Analyze plan for security and cost implications
analyze_plan_impact "$app_name"
}
```

The analyze_plan_impact helper parses the JSON plan to count resources created, updated, and deleted, highlight IAM-related changes that may require additional security review, and produce a coarse cost estimate based on resource types. These outputs are attached to notifications and approval requests.

D. Governance and Approval Workflow

Infrastructure changes flow through an explicit approval stage in the master pipeline. A dedicated stage requires manual confirmation before deployment and includes contextual metadata for reviewers:

```
stage {
 name = "ApprovalApply"
  action {
   name      = "ManualApproval"
   category  = "Approval"
   owner     = "AWS"
   provider  = "Manual"
   version   = "1"
   run_order = 3

   configuration = {
     NotificationArn    = aws_sns_topic.pipeline_notifications.arn
     ExternalEntityLink                                                                                    =
"https://console.aws.amazon.com/codesuite/codepipeline/pipelines/${local.application_name}/view"
     CustomData         = jsonencode({
       pipeline_name = local.application_name
       commit_id     = "$${codepipeline.PipelineExecutionId}"
       timestamp     = "$${codepipeline.PipelineExecutionStartTime}"
       applications  = "Auto-discovered from repository scan"
     })
   }
  }
}
```

Notifications are delivered to platform and security teams via a shared messaging topic, and additional subscriptions can be configured for finance or application owners. All approvals or rejections are logged with timestamps and approver identities to support compliance requirements.

E. Controlled Deployment and State Management

Deployment applies the approved plan with safeguards for state backup and rollback:

```
deploy_infrastructure() {
 local app_name=$1
  echo "==== Terraform Deployment for $app_name ===="
 # Pre-deployment validation
 validate_deployment_prerequisites "$app_name"
 # State backup before deployment
 backup_terraform_state "$app_name"
 # Execute approved plan
 terraform init || { echo "Terraform init failed"; exit 1; }
 # Apply with comprehensive logging
 terraform apply \
   -no-color \
   -auto-approve \
```

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004

```
  -parallelism=10 \
  tfplan 2>&1 | tee deployment.log || handle_deployment_failure "$app_name"
 # Post-deployment validation
validate_deployment_success "$app_name"
 # Update deployment metadata
record_deployment_metadata "$app_name"
}
```

The remote backend uses a shared, encrypted state bucket and lock table, with per-team keys:

```
terraform {
 backend "s3" {
  bucket         = "org-cicd-tf-state-storage"
  key            = "teams/${team_name}/terraform.tfstate"
  region         = "us-east-1"
  dynamodb_table = "terraform-state-lock"
  encrypt        = true
 }
```

If a deployment fails, the handle_deployment_failure function restores the previous state and prepares a rollback plan for review, limiting the blast radius of changes. Each application is associated with a dedicated IAM role for pipeline execution, and state isolation plus locking prevent conflicting operations on the same infrastructure stack.

VI. DESIGN RATIONALE AND EXPECTED BENEFITS

The master-pipeline design is intended to reduce manual configuration effort and improve consistency across many application pipelines. Because application discovery is driven by Terraform files in a shared repository, teams can obtain standardized pipelines by following a simple repository convention instead of filing requests for bespoke setups. Centralizing validation stages ensures that security and quality checks are systematically applied whenever infrastructure changes are proposed.

Using a single template to synthesize per-application pipelines helps maintain configuration consistency and simplifies updates: enhancements to validation, governance, or deployment behavior can be made once and applied across all pipelines. Shared state storage and locking, combined with least-privilege IAM roles, support safe parallel operation by many teams while keeping their infrastructure and state logically isolated.

While the exact quantitative impact depends on organizational context, the architecture is designed to reduce onboarding time for new pipelines, improve the uniformity of security scanning and approval practices, and allow platform engineers to focus on evolving shared templates and policies rather than constructing individual pipelines.

VII. DISCUSSION AND LIMITATIONS

The master-pipeline approach demonstrates that centrally defined templates and automated discovery can provide both governance and autonomy. However, several limitations remain. First, the design assumes Terraform-based infrastructure definitions; applications using other tools require additional integration work. Second, the implementation is specific to a single cloud provider and a particular set of CI/CD services, which limits portability to multi-cloud environments. Third, as the number of applications grows, discovery and validation stages may need additional optimization or sharding to maintain acceptable execution times.

From a security perspective, least-privilege role design and remote state isolation help reduce blast radius, but misconfiguration risks persist if templates and policies are not carefully reviewed. Governance rules must balance strictness with usability to avoid excessive approval overhead or incentives to bypass paved-road workflows. Cost estimation based on resource counts provides only an approximate view, and deeper integration with billing and usage data would be needed for precise cost management.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 2, March-April 2024||

DOI:10.15662/IJRPETM.2024.0702004

VIII. CONCLUSION

This paper presented a master-pipeline architecture for scalable management of multi-application CI/CD pipelines. By automatically discovering Terraform-based applications in a shared repository and provisioning standardized pipelines per application, the system enables self-service onboarding while preserving centralized governance and security controls. The implementation supports multiple source providers, integrates multi-tool validation and security scanning, and incorporates structured approval workflows, impact analysis, and controlled deployment with rollback.

The pattern illustrates how organizations can achieve both centralized control and team autonomy by treating pipelines themselves as provisioned infrastructure and automating their lifecycle through a master controller. Future work includes extending the architecture to multi-account and multi-region deployments, integrating more accurate cost and performance analytics, exploring AI-assisted optimization of pipeline parameters, and generalizing the design to additional orchestration and infrastructure-as-code tools.

REFERENCES

CI/CD & DevOps Foundations

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.
https://continuousdelivery.com/

[2] GitOps Working Group, "GitOps Principles," OpenGitOps.
https://opengitops.dev/

[3] Amazon Web Services, "AWS CodePipeline User Guide."
https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html

[4] Amazon Web Services, "AWS CodeBuild User Guide."
https://docs.aws.amazon.com/codebuild/latest/userguide/welcome.html

[5] Amazon Web Services, "AWS CodeStar Connections."
https://docs.aws.amazon.com/dtconsole/latest/userguide/connections.html

[6] Amazon Web Services, "AWS CodeCommit User Guide."
https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html

[7] TFLint Project, "TFLint Documentation."
https://github.com/terraform-linters/tflint

[8] Bridgecrew, "Checkov Documentation."
https://www.checkov.io/

[9] Aqua Security, "TFSec Documentation."
https://aquasecurity.github.io/tfsec/

[10] Amazon Web Services, "AWS Identity and Access Management (IAM)."
https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html