International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P105
Eureka Vision Publication | Volume 7, Issue 1, 23-29, 2026

Original Article

Developing End-to-End Concourse CI/CD Pipelines with Automated Testing, Scanning, Canary Deployments, and Rollback Logic

Sneha Palvai[1], Vivek Jain[2]
[1]DevOps/AWS Engineer, Comcast, Philadelphia, USA.
[2]Digital Development Manager, Academy Sports Plus Outdoors, Texas, USA.

Received On: 23/11/2025 ***Revised On: 24/12/2025*** ***Accepted On: 01/01/2026*** ***Published On: 14/01/2026***

Abstract - The increasing demand for rapid software delivery has elevated Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines into mission-critical systems. Modern pipelines must not only automate builds and deployments but also ensure software quality, security, reliability, and compliance. This paper presents a comprehensive end-to-end approach for designing and implementing CI/CD pipelines using Concourse CI, integrating automated testing, security scanning, progressive canary deployments, and automated rollback mechanisms. A reference architecture and reusable pipeline patterns are proposed, followed by three practical case studies across cloud-native microservices, regulated enterprise platforms, and data engineering pipelines. The paper further evaluates pipeline effectiveness using industry-standard metrics and explores future directions including policy-as-code, software supply chain security, SBOM-driven delivery, and AI-assisted continuous testing.

Keywords - CI/CD, Concourse CI, DevSecOps, Continuous Testing, Canary Deployment, Rollback Automation, Kubernetes, Software Supply Chain.

1. Introduction

Continuous delivery has become a foundational capability for modern software-driven organizations. High-performing teams deploy changes frequently while maintaining system reliability and security. However, increased deployment velocity introduces significant operational risk if validation, security, and release controls are insufficient. Research in Site Reliability Engineering (SRE) emphasizes that reliability must be designed into systems through automation, observability, and controlled release strategies rather than relying on manual intervention [1].

Concourse CI is an open-source CI/CD platform that emphasizes declarative pipelines, containerized execution, and explicit modeling of external state [2]. These characteristics make it well-suited for building reproducible, auditable, and scalable delivery pipelines. While Concourse is often adopted for build automation, its design enables the construction of fully automated delivery systems that include testing, security scanning, progressive deployment, and rollback logic.

This paper presents an end-to-end CI/CD architecture using Concourse CI and makes the following contributions:

1. A reference architecture for secure and reliable CI/CD pipelines.
2. Pipeline design patterns for automated testing, DevSecOps integration, canary deployments, and rollback.
3. Real-world-inspired case studies demonstrating practical adoption.
4. A discussion of future trends shaping next-generation CI/CD systems.

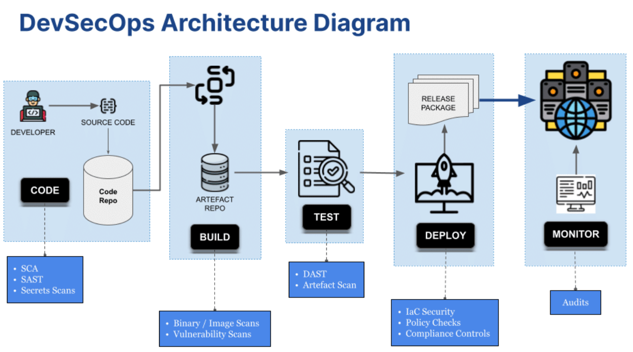


Figure 1. DevSecOps Architecture Diagram

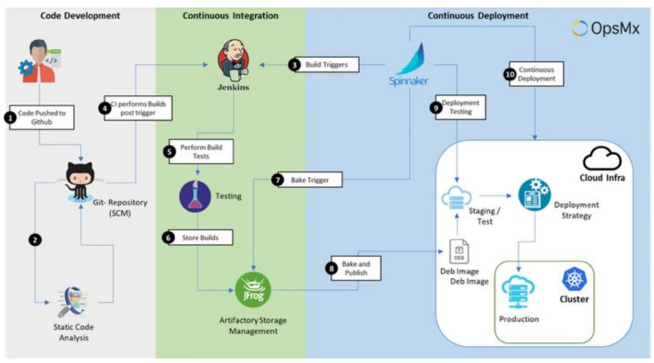


Figure 2. CI/CD Pipeline

2. Background and Motivation

2.1. Evolution of CI/CD Pipelines

Traditional CI systems focused primarily on code compilation and unit testing. Over time, CI/CD pipelines have expanded to include infrastructure provisioning, security validation, and production deployment automation [3]. The adoption of containers and Kubernetes further accelerated this shift toward immutable artifacts and declarative environments [4].

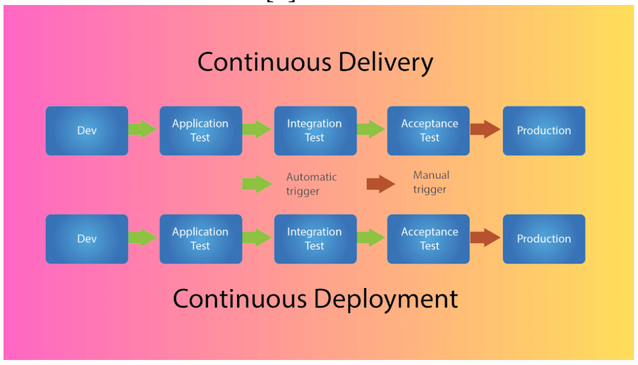


Figure 3. Continuous Delivery Vs Continuous Deployment

2.2. Continuous Testing as a Core Capability

Continuous testing ensures that quality checks are embedded across the entire delivery lifecycle rather than isolated to a single stage. Empirical studies show that integrating automated testing at every pipeline phase reduces defect leakage, accelerates feedback, and improves release confidence [6], [15]. Continuous testing is therefore a prerequisite for safe high-frequency deployments.

2.3. Motivation for Concourse CI

Concourse CI enforces a strict separation between pipeline definition and execution, using ephemeral containers for all tasks [2]. This model reduces configuration drift, improves reproducibility, and supports strong governance and audit requirements. These properties make Concourse particularly suitable for regulated and large-scale environments.

3. Reference Architecture

3.1. End-to-End Pipeline Stages

The proposed CI/CD pipeline consists of the following stages:

1. Source and Build: Source code checkout, static analysis, unit testing, and artifact creation.
2. Verification: Integration tests, contract tests, and performance smoke tests.
3. Security and Compliance: SAST, SCA, container image scanning, secret detection, and IaC validation.
4. Artifact Hardening: Image signing, Software Bill of Materials (SBOM) generation, and metadata attachment.
5. Progressive Deployment: Canary deployment with incremental traffic shifting.
6. Observation and Decision: Runtime telemetry evaluation to determine promotion or rollback.
7. Post-Deployment: Release tagging, notifications, audit logging, and metrics collection.

This layered architecture aligns with DevSecOps and secure software delivery frameworks [5], [9].

3.2. Key System Components

- Version Control: Git-based repositories for application and infrastructure code.
- Artifact Registry: OCI-compliant container registries.
- Security Toolchain: SAST, SCA, container, and IaC scanning tools.
- Runtime Platform: Kubernetes with ingress or service mesh support.
- Observability Stack: Metrics, logs, and traces (e.g., Prometheus-based monitoring).
- Progressive Delivery Controller: Canary management using rollout controllers.

4. Pipeline Design Patterns

Modern CI/CD systems must balance **speed, safety, and reliability**. The following pipeline design patterns illustrate how Concourse CI enables high-confidence continuous delivery through declarative automation, progressive validation, and closed-loop feedback.

4.1. Immutable Artifact Promotion

Artifacts are built once and promoted across environments using immutable identifiers such as digests or commit hashes. This approach eliminates environment-specific builds and improves traceability and auditability [4], [10].

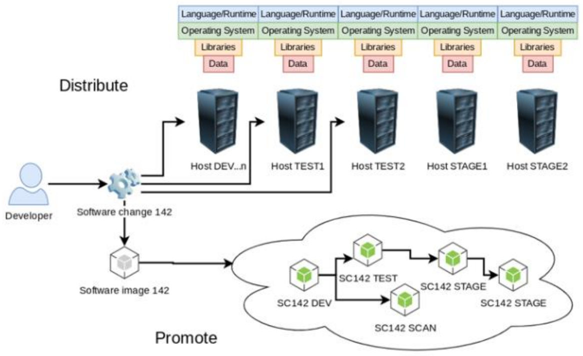


Figure 4. Immutable Infrastructure CI/CD

4.1.1. Key Characteristics

- **Single source of truth** for artifacts
- Artifact identity is immutable and verifiable
- Promotion is a metadata operation, not a rebuild

4.1.2. Benefits

- Eliminates configuration drift between environments

- Improves traceability and auditability**,** as every deployment can be traced back to a specific commit and build [4], [10]
- Enables deterministic rollback by redeploying a known-good artifact

4.1.3. Concourse Implementation

- Artifact produced once (e.g., Docker image)
- Stored in a registry with immutable tags or digests
- Promotion jobs reference the same artifact across all environments

4.2. Automated Testing Pyramid

The pipeline enforces a structured testing strategy:

- Unit tests executed on every commit.
- Integration and contract tests executed before environment promotion.
- End-to-end tests executed during canary or post-deployment phases.

Research indicates that such continuous testing strategies significantly improve delivery outcomes and system stability [6], [15].

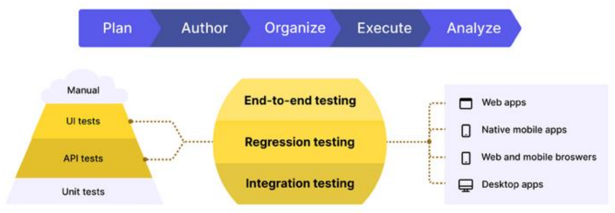


Figure 5. Testing Phases

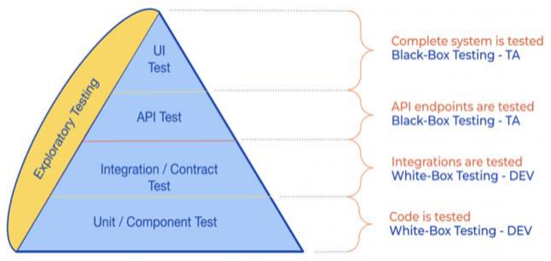


Figure 6. Testing Pyramid

4.3. DevSecOps Gates and Policy-as-Code

Security controls are embedded directly into the pipeline using automated gates. Policies define acceptable vulnerability thresholds and configuration rules. Policy-as-code ensures consistency, versioning, and auditability across environments [11].

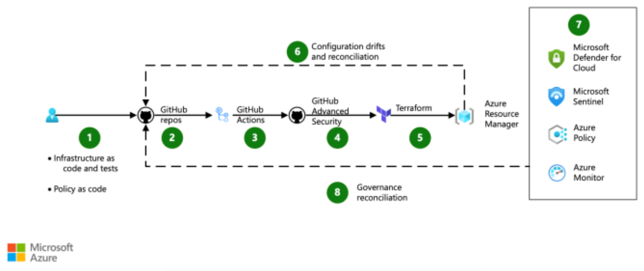


Figure 7. DevSecOps for IAC

4.4. Canary Deployment and Rollback Automation

Canary deployments gradually expose a new version to production traffic while monitoring key Service Level Indicators (SLIs) such as error rate and latency [7]. If thresholds are exceeded, the pipeline automatically triggers a rollback, restoring the previous stable version and rerouting traffic [8].

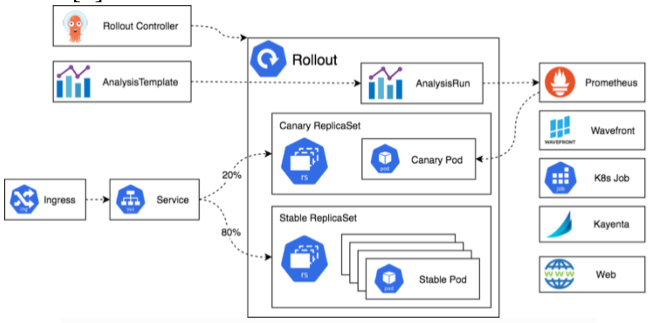


Figure 8. Progressive-Delivery-Rollouts-Analysis

5. Case Studies

This section presents real-world-inspired case studies demonstrating how the proposed Concourse-based CI/CD architecture and pipeline design patterns are applied across diverse operational contexts. Each case highlights measurable improvements in reliability, security, and recovery, aligning with established SRE and DevSecOps principles.

5.1. Cloud-Native Microservices Platform

5.1.1. Context

A large-scale retail platform operated a **cloud-native microservices architecture** deployed on Kubernetes. The system comprised dozens of independently deployable services supporting high-traffic e-commerce workflows. Frequent releases were necessary to support rapid feature experimentation and seasonal demand spikes.

5.1.2. Pipeline Implementation

The organization implemented Concourse-based CI/CD pipelines with the following characteristics:

- Immutable artifact creation and promotion across environments
- Automated testing and security scans at each promotion stage
- Canary deployments using Kubernetes-native traffic routing
- Continuous monitoring of real-time metrics during rollout

5.1.3. Observability and Control

During canary rollout, the pipeline evaluated key **Service Level Indicators (SLIs)**:

- HTTP error rates
- Request latency percentiles
- Pod health and restart frequency

Promotion decisions were fully automated. If SLI thresholds were exceeded, rollback jobs were triggered immediately.

5.1.4. Outcomes

- Reduced change failure rate by limiting the blast radius of releases
- Improved Mean Time to Recovery (MTTR) through automated rollback
- Enabled higher deployment frequency without compromising stability

These outcomes align closely with SRE best practices emphasizing automation, observability, and controlled release strategies [1], [7].

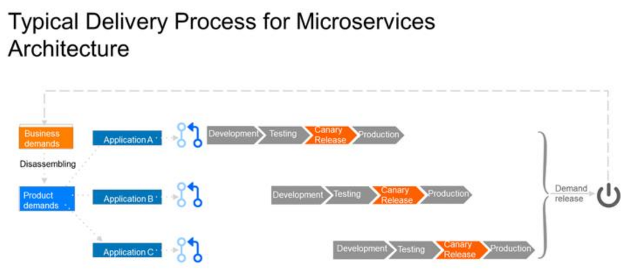


Figure 9. Typical Delivery Process for Microservices Architecture

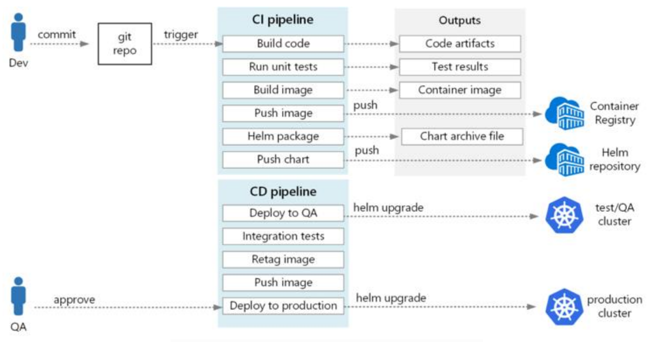


Figure 10. AKS CI/CD Flow

5.2. Regulated Enterprise Environment

5.2.1. Context

A highly regulated enterprise environment—subject to stringent compliance and audit requirements—required strong guarantees around software provenance, integrity, and security controls. Manual approvals and ad hoc security checks had historically slowed delivery and increased audit overhead.

5.2.2. Pipeline Implementation

The organization adopted Concourse pipelines with **security and compliance embedded as code**, including:

- Mandatory static and dependency security scans
- Cryptographic artifact signing during build
- Promotion-by-digest to ensure artifact immutability
- Automated policy evaluation at each pipeline stage

Security policies were version-controlled and enforced uniformly across all environments.

5.2.3. Compliance Alignment

The pipeline design aligned with **secure software supply chain** and **NIST secure software development guidelines**, ensuring:

- Full traceability from source commit to production deployment
- Deterministic and repeatable deployments
- Automatically generated audit artifacts

5.2.4. Outcomes

- Simplified audit preparation through built-in evidence generation
- Reduced reliance on manual approvals without weakening controls
- Improved consistency and confidence in production releases [9], [10]

This case demonstrates how automation can strengthen compliance rather than undermine it.

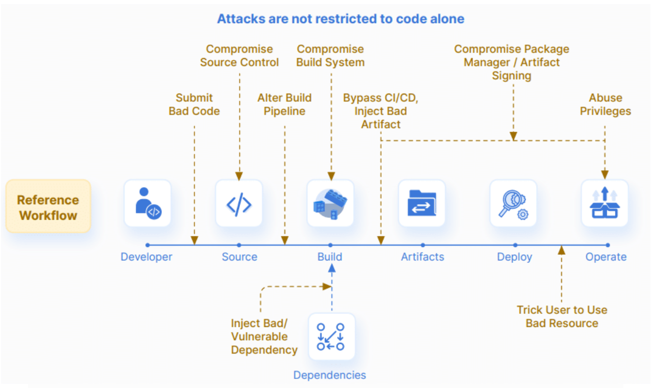

Attacks are not restricted to code alone
Compromise Source Control
Compromise Build System
Compromise Package Manager / Artifact Signing
Submit Bad Code
Alter Build Pipeline
Bypass CI/CD, Inject Bad Artifact
Abuse Privileges
Reference Workflow
Developer
Source
Build
Artifacts
Deploy
Operate
Inject Bad/ Vulnerable Dependency
Dependencies
Trick User to Use Bad Resource

Figure 11. Secure and Compliant CI/CD Pipeline with Artifact Signing and Policy Enforcement.

5.3. Data Engineering and ETL Pipelines

5.3.1. Context

A large-scale data platform managed complex **ETL pipelines** responsible for analytics and downstream business reporting. Failures in production data pipelines posed a high risk of **data corruption and downstream inaccuracies**, making traditional rollback approaches impractical.

5.3.2. Pipeline Implementation

The platform implemented staged promotion using Concourse pipelines:

- Data transformations were validated on **sa**mpled datasets

- Canary processing was applied to subsets of production data
- Pipeline stages were fully observable with data-quality metrics

Instead of reverting state, rollback was implemented as a forward-fix strategy, replaying data with corrected logic.

5.3.3. Rollback and Resilience Strategy

Key mechanisms included:

- Idempotent data processing tasks
- Controlled replay of historical datasets
- Automated detection of schema and quality violations

5.3.4. Outcomes

- Reduced risk of large-scale data corruption
- Improved pipeline resilience and recovery speed
- Increased confidence in production data releases [12]

This case highlights that CI/CD principles—when adapted correctly—extend beyond application delivery to data engineering systems.

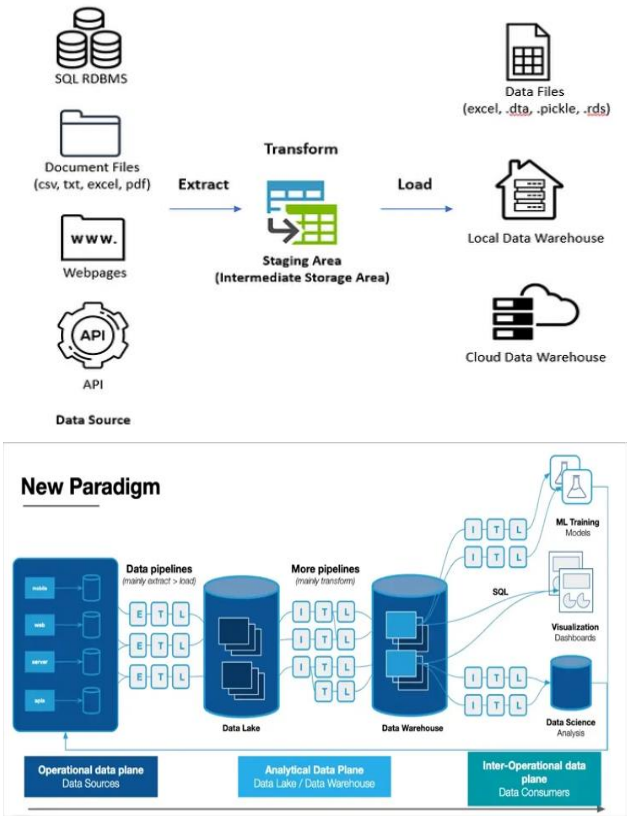

SQL RDBMS
Document Files (csv, txt, excel, pdf)
Webpages
API
Data Source
Extract
Transform
Staging Area (Intermediate Storage Area)
Load
Data Files (excel, .dta, .pickle, .rds)
Local Data Warehouse
Cloud Data Warehouse
New Paradigm
Data pipelines (mainly extract > load)
More pipelines (mainly transform)
Data Lake
Data Warehouse
SQL
ML Training Models
Visualization Dashboards
Data Science Analysis
Operational data plane Data Sources
Analytical Data Plane Data Lake / Data Warehouse
Inter-Operational data plane Data Consumers

Figure 12. Stage promotion and rollback strategy for data engineering and ETL pipelines

6. Future Trends in it budgeting

Future CI/CD systems are evolving beyond automation efficiency toward trust, intelligence, and governance by design. As software delivery becomes increasingly distributed and security-critical, pipelines themselves are emerging as first-class, policy-governed systems.

6.1.1. Software Supply Chain Security and SLSA Provenance

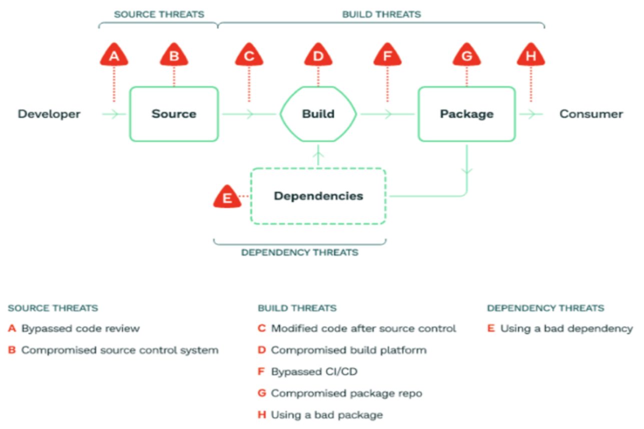


Figure 13. Software Supply Chain Security Using SLSA-Compliant Provenance And Attestations

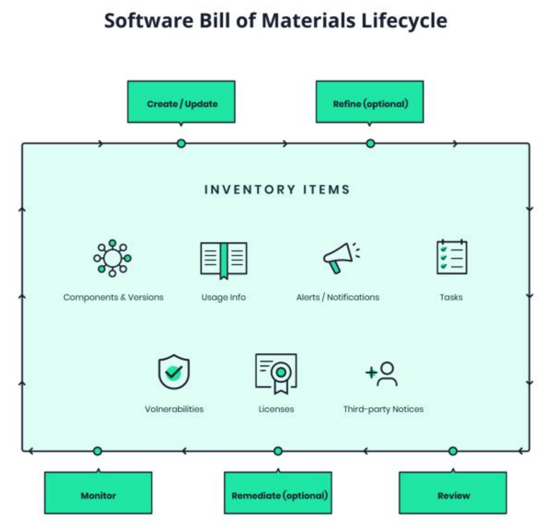
Modern attacks increasingly target the software supply chain, exploiting weaknesses in build systems, dependencies, and artifact repositories. To mitigate these risks, future pipelines will adopt SLSA-compliant (Supply-chain Levels for Software Artifacts) provenance models.

Key characteristics include:

- Cryptographically verifiable build provenance
- Non-falsifiable attestations linking source, build, and artifact
- Tamper-resistant metadata stored alongside artifacts

SLSA-aligned pipelines ensure that every deployed artifact can be traced back to a trusted build process, significantly reducing the risk of dependency poisoning and unauthorized modifications [10].

6.1.2. SBOM-First Delivery Models



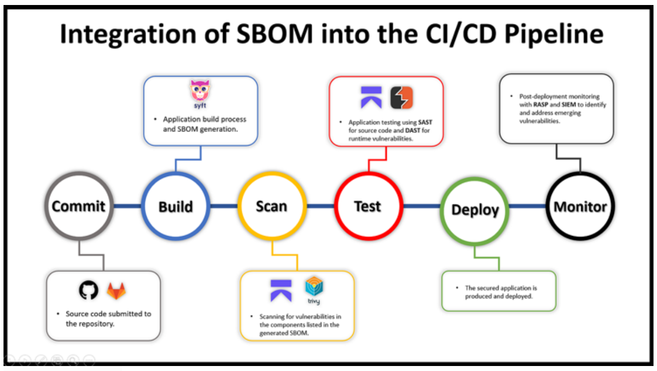

Integration of SBOM into the CI/CD Pipeline
Commit
Build
Scan
Test
Deploy
Monitor
syft
Application build process and SBOM generation.
Application testing using SAST for source code and DAST for runtime vulnerabilities.
Post-deployment monitoring with RASP and SIEM to identify and address emerging vulnerabilities.
Source code submitted to the repository.
trivy
Scanning for vulnerabilities in the components listed in the generated SBOM.
The secured application is produced and deployed.

Figure 14. SBOM-First CI/CD Delivery Pipeline Enabling Vulnerability Transparency

Future CI/CD systems will treat **Software Bills of Materials (SBOMs)** as mandatory deployment artifacts rather than optional reports. SBOM-first delivery enables organizations to understand and manage risk across their entire dependency graph.

Emerging practices include:

- Automatic SBOM generation during build stages
- Continuous vulnerability assessment against deployed SBOMs
- Compliance reporting integrated directly into release pipelines

This approach improves vulnerability transparency**,** accelerates incident response, and aligns with regulatory expectations in regulated and enterprise environments [9].

6.1.3. Policy-as-Code Standardization Across Governance Domains

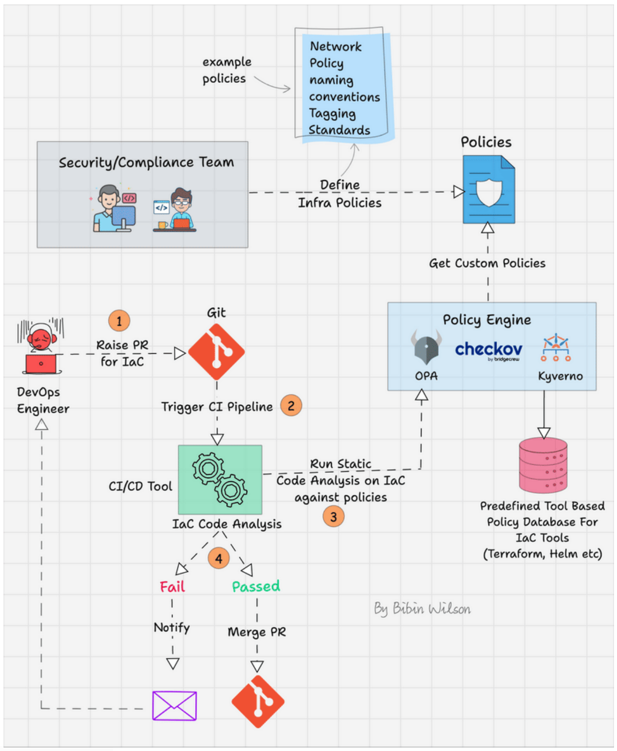


Figure 15. Policy-as-Code Enforcement Across Security, Compliance, And Governance Domains

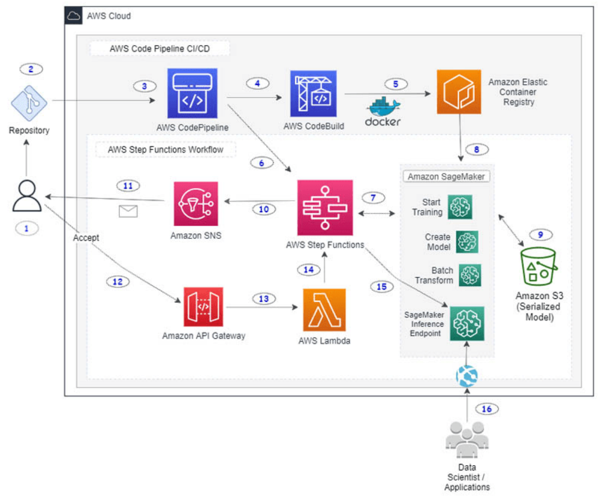
As delivery complexity grows, organizations are converging on **policy-as-code** as a unifying mechanism for enforcing security, compliance, and operational governance.

Future pipelines will standardize:

- Security policies (vulnerability thresholds, secrets handling)
- Infrastructure governance (networking, IAM, encryption)
- Release and operational controls (promotion rules, blast-radius limits)

By codifying policies and versioning them alongside application code, CI/CD pipelines become **self-governing systems**, ensuring consistency, auditability, and cross-team alignment [11].

6.1.4. AI-Assisted Continuous Testing and Pipeline Optimization



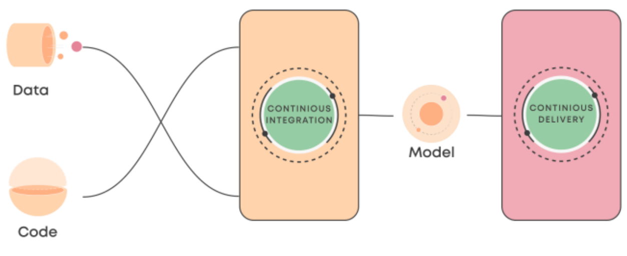


Figure 15. AI-Assisted Continuous Testing and Adaptive Pipeline Optimization

Artificial intelligence and machine learning are poised to transform CI/CD pipelines from static workflows into adaptive, self-optimizing systems**.**

Emerging AI-driven capabilities include:

- Intelligent test selection based on code-change impact
- Failure prediction using historical pipeline telemetry

- Dynamic optimization of pipeline execution paths

These techniques reduce test execution time while improving defect detection rates, enabling faster feedback loops without sacrificing quality. Research indicates that AI-assisted continuous testing significantly enhances delivery performance and system reliability [14], [15].

6.2. Synthesis: The Next Generation of CI/CD

Collectively, these trends signal a shift toward CI/CD systems that are:

- Trust-aware (secure by provenance and attestation)
- Transparent (SBOM-driven visibility)
- Governed (policy-as-code enforcement)
- Intelligent (AI-assisted decision-making)

Such systems align closely with SRE principles by embedding reliability, security, and resilience directly into the delivery lifecycle rather than treating them as external concerns.

7. Conclusion

This paper demonstrates how Concourse CI can be used to design and implement robust, end-to-end CI/CD pipelines that integrate automated testing, security scanning, progressive canary deployments, and automated rollback logic. By combining immutable artifact promotion, continuous testing, policy-driven security and governance gates, and telemetry-based decision-making, organizations can safely increase deployment velocity while significantly reducing operational risk. The proposed reference architecture and real-world-inspired case studies illustrate that reliable continuous delivery is achievable at scale when automation, observability, and governance are treated as first-class pipeline concerns rather than post-deployment safeguards. These results reinforce SRE principles that reliability must be engineered into delivery systems through declarative workflows, measurable signals, and controlled release strategies. As CI/CD systems continue to evolve toward supply chain security, policy standardization, and AI-assisted optimization, platforms such as Concourse CI provide a strong foundation for building secure, auditable, and resilient delivery pipelines capable of supporting modern software-driven enterprises.

References

[1] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*, O'Reilly Media, 2016.

[2] Concourse CI, "Concourse Documentation," https://concourse-ci.org

[3] J. Humble and D. Farley, *Continuous Delivery*, Addison-Wesley, 2011.

[4] Kubernetes Authors, "Kubernetes Documentation," https://kubernetes.io

[5] OWASP Foundation, "OWASP Software Assurance Maturity Model (SAMM)," 2020.

[6] L. Crispin and J. Gregory, *Agile Testing*, Addison-Wesley, 2009.

[7] Argo Project, "Argo Rollouts: Progressive Delivery for Kubernetes," https://argo-rollouts.readthedocs.io

[8] M. Fowler, "Blue-Green Deployment," martinfowler.com, 2010.

[9] [NIST, *Secure Software Development Framework (SSDF)*, NIST SP 800-218, 2022.

[10] OpenSSF, "Supply-chain Levels for Software Artifacts (SLSA)," https://slsa.dev

[11] Open Policy Agent, "Policy-as-Code," https://www.openpolicyagent.org

[12] G. Dehghani, *Data Mesh*, O'Reilly Media, 2022.

[13] N. Forsgren, J. Humble, and G. Kim, *Accelerate*, IT Revolution Press, 2018.

[14] Google Research, "Machine Learning for Systems and Systems for Machine Learning," 2020.

[15] V. Jain, "Continuous Testing in CI/CD Pipelines," *International Journal of Innovative Research and Creative Technology*, vol. 9, no. 1, pp. 1–7, 2023, doi: 10.5281/zenodo.14883221.