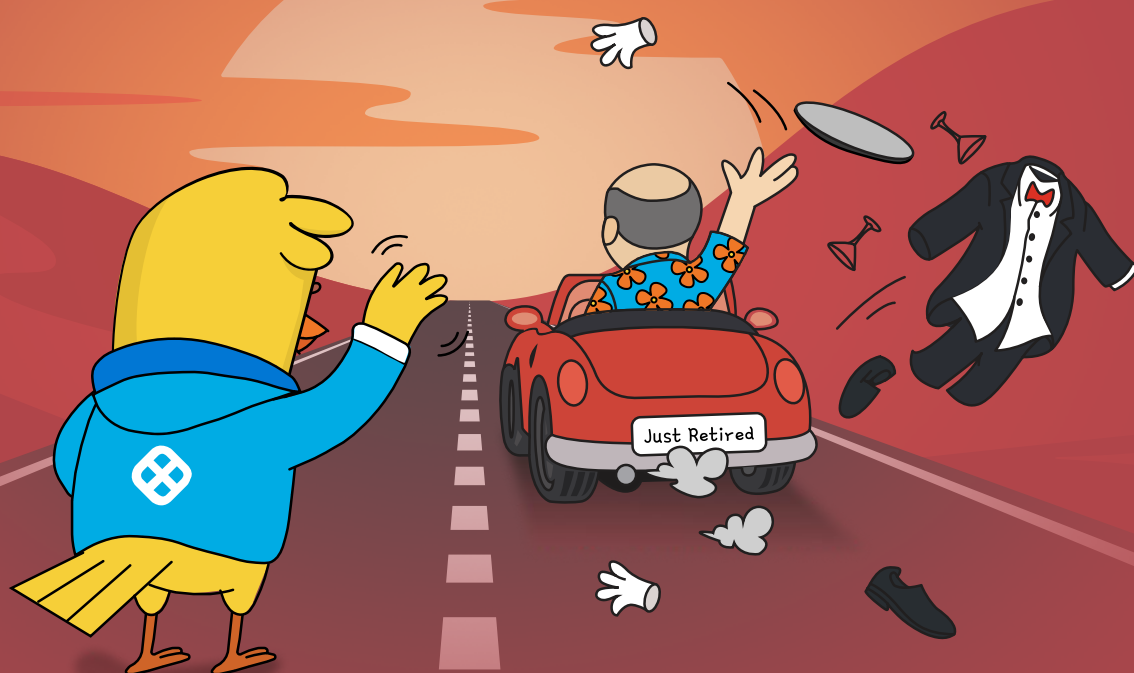




Migrating off Jenkins to a Modern CI/CD Solution

Break free from CI/CD complexity and
embrace AI for Software Delivery



Executive Summary

For many organizations over the past decade, Jenkins has been the cornerstone of continuous integration and continuous delivery (CI/CD). However, as DevOps practices evolve and cloud-native technologies become standard, many teams find that Jenkins struggles to keep up with modern requirements. This white paper examines why Jenkins gained such widespread adoption, the challenges organizations face, why migration can be difficult, and a structured approach to moving to Harness, an AI for Software Delivery platform designed to address these pain points. The benefits of migration include reduced maintenance overhead, faster and more secure builds, ease of deployments in various environments, enhanced developer experience, and advanced features like AI-powered verification and rollback that are critical in today's competitive landscape.

Introduction

The Evolution of CI/CD

Continuous Integration and Continuous Delivery have transformed from novel concepts to essential practices in modern software development. As organizations strive for faster releases, higher quality, and increased automation, the tools that support these practices must evolve accordingly. Jenkins emerged as an early leader in this space, providing a flexible platform for automation that helped countless organizations take their first steps toward CI/CD.

However, as software delivery becomes more complex, with microservices architectures, containerization, and cloud-native approaches becoming standard, many teams find that Jenkins creates more friction than flow in their delivery pipelines. This realization has sparked a migration trend toward more modern CI/CD solutions like Harness that are purpose-built for today's development challenges.

Why Jenkins Has Been a Popular Choice

Widespread Adoption and Community Support

Jenkins has amassed well over millions of developers worldwide¹, making it one of the industry's most widely used CI/CD tools. This widespread adoption means finding talent familiar with Jenkins is relatively easy, reducing onboarding time for new team members.

Open-Source Foundation

As an open-source solution, Jenkins offers organizations a free entry point into CI/CD automation. This has made it particularly attractive for teams looking to implement CI/CD practices without significant upfront investment.

Flexibility and Extensibility

Jenkins was designed with extensibility in mind, offering a plugin architecture that allows users to adapt the tool to almost any workflow or technology stack. With nearly 2,000 community-developed plugins available, users can find extensions for virtually any use case imaginable.

Developer-Centric Design

Created by Kohsuke Kawaguchi² while working at Sun Microsystems, Jenkins was built by a developer facing the frustrations of manual build and test cycles. This developer-centric approach resonated with engineering teams who saw Jenkins as a tool built to solve real problems they encountered daily.

Distributed Architecture

Jenkins leverages a controller-agent architecture³ that allows organizations to distribute workloads across multiple nodes. This design enables teams to run builds and tests in parallel across different environments and operating systems, improving throughput and efficiency.

¹ <https://appvibe.com/blog/why-is-jenkins-still-so-popular-in-2023/>

² <https://www.jenkins.io/blog/authors/kohsuke/>

³ <https://devopscube.com/jenkins-architecture-explained/>

Ease of Installation and Setup

For basic use cases, Jenkins is relatively straightforward to install and configure. It can run on any system with a Java Runtime Environment (JRE), be deployed in containers, or installed through native system packages, giving teams flexibility in how they deploy it.

Pain Points of Jenkins in Today's DevOps Landscape

The Plugin Nightmare

While Jenkins' extensive plugin ecosystem is often cited as a strength, it has increasingly become a source of frustration and technical debt for many organizations:



Not Built for Continuous Delivery/Deployment: Teams often rely on plugins to build out their CD workflows, which creates a lot of operational overhead and furthers the plugin nightmare.



Dependency Hell: The interdependency between plugins creates complex compatibility issues that make upgrades risky and time-consuming.



Outdated and Unmaintained Plugins: Many plugins are poorly maintained or completely abandoned, leading to security vulnerabilities and compatibility problems with newer Jenkins versions.



Documentation Gaps: Many plugins lack comprehensive documentation, making troubleshooting and proper implementation challenging.

Resource Intensity and Scalability Issues

Jenkins installations often become resource-hungry as they scale:



Excessive Resource Consumption: Jenkins setups frequently demand excessive RAM and CPU resources, straining infrastructure and limiting scalability.



Performance Degradation: As the number of jobs increases, the Jenkins server can become a bottleneck, slowing down the entire CI/CD process.

Maintenance Burden

The ongoing maintenance of Jenkins represents a significant overhead for many organizations:



Dedicated Personnel Required: Maintaining Jenkins often necessitates a dedicated team, pulling valuable resources away from product development.



Configuration Drift: Over time, Jenkins instances tend to accumulate manual configurations that are difficult to track and reproduce, creating "snowflake" servers that become impossible to recreate if lost.

Limited Infrastructure-as-Code (IaC) Support

Modern DevOps practices emphasize defining infrastructure and configuration as code, an area where Jenkins has traditionally fallen short:



Manual Configuration Prevalence: Jenkins makes it too easy to create configurations through clicking around the UI rather than defining them as code.



Rudimentary IaC Support: While improving, Jenkins' support for infrastructure-as-code and configuration-as-code remains less mature than purpose-built modern solutions.

Containerization/Modern Environment Challenges

As organizations move toward containerized workloads, Jenkins' architecture presents obstacles:



Difficult to Containerize: Jenkins was not originally designed with containerization in mind, making it challenging to run in modern container orchestration platforms like Kubernetes.



Ephemeral Environment Limitations: Jenkins struggles with truly ephemeral build environments, a key requirement for modern cloud-native applications.

User Experience Issues

Jenkins' user interface and user experience have not kept pace with modern expectations:



Outdated UI: Jenkins' interface is frequently criticized as outdated and unintuitive, making it less user-friendly than modern alternatives.



Complex Pipeline Syntax: The Jenkinsfile syntax is considered overly complex and difficult to master compared to competitors' offerings.

Why Migrating off of Jenkins is Tedious

Substantial Investment in Existing Pipelines

Many organizations have invested years of development effort into their Jenkins pipelines:

Pipeline Sprawl: Large enterprises often have thousands of Jenkins pipelines accumulated over many years.

Custom Code and Shared Libraries: Organizations typically develop extensive custom code and shared libraries specific to their Jenkins implementation.

Institutional Knowledge and Documentation

Migration requires addressing a significant body of institutional knowledge tied to Jenkins:

Internal Documentation: Organizations often have extensive internal documentation centered around Jenkins workflows and troubleshooting.

Team Expertise: Teams have developed specialized knowledge about their Jenkins setup that doesn't translate directly to new platforms.

Business Case Challenges

Justifying the migration from a financial and operational perspective can be difficult:

Developer Time Investment: Migration requires substantial developer time that could otherwise be spent on product features or improvements.

Opportunity Cost: The resources required for migration represent a significant opportunity cost, often approaching "close to a million or more in developer time".

New Issues Post-Migration: Like any major system rewrite, migration introduces a new set of issues that need to be addressed.

Technical Complexity

The technical aspects of migration present numerous challenges:

Plugin Dependency Analysis: Understanding what plugins are used, their exact functions, and their interdependencies is a complex undertaking.

Agent Configuration Understanding: Determining what's installed on Jenkins agents and how they're configured requires detailed analysis.

Authentication and Security Migration: Moving authentication methods, SSH keys, API tokens, and other secrets requires careful planning to maintain security.

What to Jenkins Users Should Know About Harness

Organizations coming from Jenkins often bring years of accumulated patterns, tooling habits, and mental models around how CI/CD should work. While Harness introduces a different approach to automation and platform experience, many foundational ideas will feel familiar. Concepts like pipelines, stages, steps, triggers, secrets, and agents still exist - what changes is how these are implemented, maintained, and secured.

This section outlines what engineers and platform teams should expect when transitioning from Jenkins to Harness. It highlights where the tools align conceptually and where Harness intentionally diverges to offer greater reliability, governance, and developer experience.

Pipeline Creation and Execution

In Jenkins, pipelines are defined using scripted or declarative Groovy in Jenkinsfiles. Harness pipelines can also be written as code using YAML, but there's also a visual editor that gives teams flexibility in authoring and reviewing workflows. Jenkins pipelines are often monolithic and tied to repository structure, while Harness encourages modular, reusable pipeline components. Harness also uses AI to generate pipelines. The AI doesn't just create generic pipelines; it leverages your existing templates, tool configurations, environments, and governance policies to ensure each pipeline meets your internal standards.

One key similarity is that both tools define pipelines as a sequence of steps or stages that execute on a worker. But where Jenkins relies heavily on plugins and custom scripting, including the full use of Groovy as a general-purpose language, that flexibility often comes at the cost of consistency, maintainability, and onboarding speed. Harness takes a different approach with YAML-based pipelines that prioritize clarity and standardization without giving up power. It provides guardrails, reusable templates, custom steps, and intelligent input sets to express complex workflows in a predictable and shareable way. For use cases that require dynamic behavior, Harness supports extensibility through custom modules and integrations while still preserving the simplicity and auditability of declarative pipelines.

Plugins vs Platform Features

In Jenkins, much of the functionality is built through community plugins, whether it's Docker builds, GitHub integrations, or test reports. While this offers flexibility, it often comes with versioning conflicts, plugin maintenance overhead, and unpredictable behaviors when plugins are outdated or poorly maintained.

Harness takes a different approach by offering a built-in step library with dozens of the most commonly used CI/CD actions, such as building and pushing Docker images, running tests, scanning for vulnerabilities, and more. These steps work out of the box without requiring additional installation or configuration.

Beyond steps, Harness also introduces templates, a powerful way to define reusable logic and pipeline components. You can create templates for steps, stages, or entire pipelines, and then link or share them across teams. This makes it easy to standardize best practices, enforce security and compliance rules, and reduce duplication. For example, a team can define a single security scan stage as a template, then reuse it across dozens of pipelines without rewriting or copy-pasting YAML.

Templates enhance developer productivity, reduce onboarding time, and bring structure to what might otherwise become fragmented or inconsistent pipeline code. For example, a team can define a security scan stage as a template once and reuse it across multiple pipelines to ensure consistency and compliance without duplicating YAML. Below is a sample pipeline template to illustrate how teams can encapsulate reusable logic and share it across projects or teams.

Create New Pipeline Template

Name ⓘ
Id ⓘ : Quickstart ✎

Quickstart


Description (optional) ✎

Tags (optional) ✎

Version Label ⓘ

v1

Logo (optional)





ⓘ Max file size: 30 KB. Recommended logo size: 32 × 32 pixels.

Save To

Project

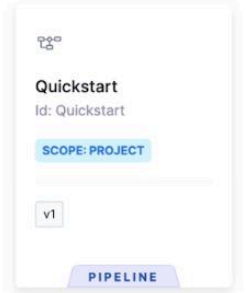
How do you want to set up your template?


Inline
Template is stored in Harness


Remote
Template is stored in your Git repository

Start

Cancel



For teams that want plugin-like extensibility, Harness CI supports custom steps and integrates with the Drone Plugins ecosystem. You can use pre-built plugins from the [Drone Plugins Marketplace](#). You can also write your own plugins and invoke them as part of your pipelines, similar to Jenkins shared libraries or custom steps, but with more structure and maintainability.

The flexibility of plugins is still there, but Harness wraps it with support, governance, and a curated experience that reduces the friction developers often face in Jenkins-based setups.

Secrets and Credentials

Both Jenkins and Harness support injecting secrets into builds, but the management approach differs. Jenkins stores credentials internally or uses plugins to fetch from secret managers. Harness provides [a native secrets manager](#) with support for [Vault](#), [AWS Secrets Manager](#), [GCP Secret Manager](#), and others, with built-in RBAC and audit logging.

The idea of securing tokens, passwords, or SSH keys remains the same, but Harness adds structure and visibility around who can access what and when.

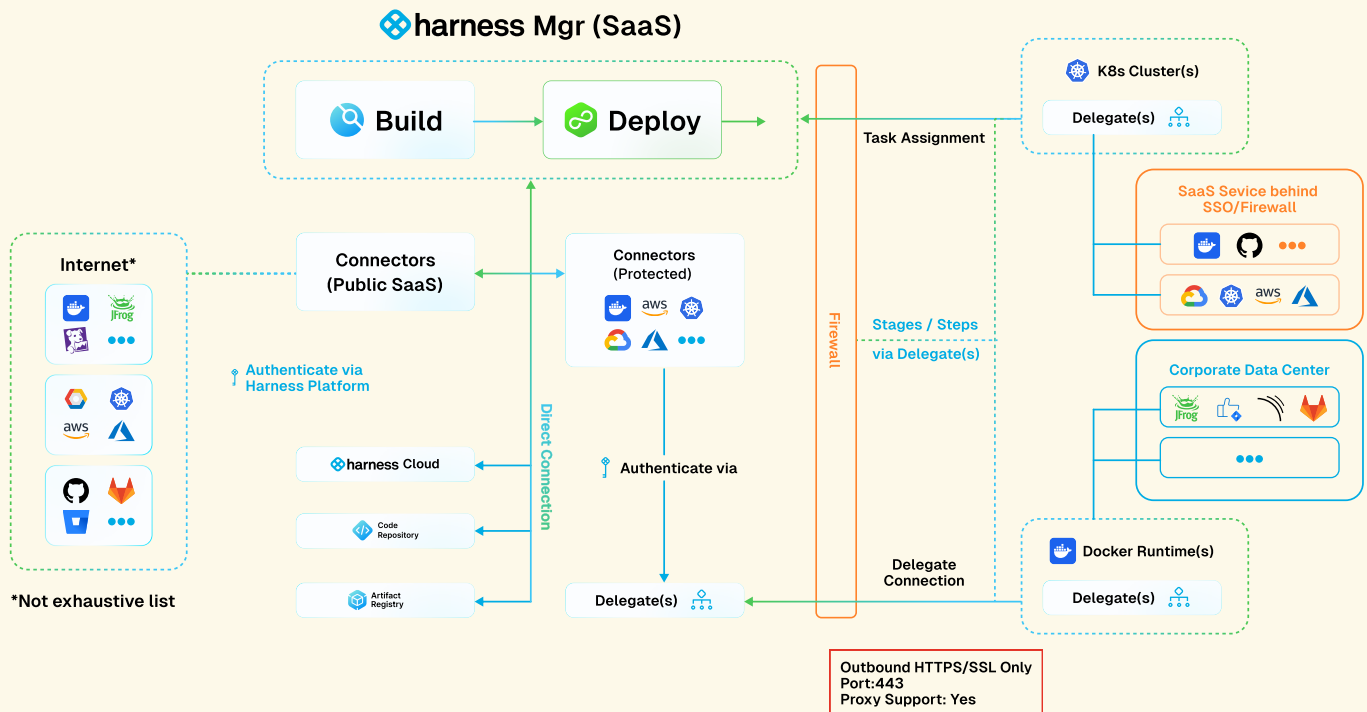
Agents and Infrastructure

In Jenkins, builds are executed by agents (or nodes) that you configure and manage yourself. These agents can be static virtual machines, Docker containers, or dynamic Kubernetes pods, but it's up to the user to provision, secure, scale, and maintain them. Many teams build their own fleet of agents and use labels to assign jobs across them, often via plugins like the Kubernetes plugin or EC2 plugin.

Harness takes a similar conceptual approach - you still need some kind of executor, but it introduces more structure and flexibility. You can choose from several execution infrastructures: Kubernetes (via Harness-managed pods), Docker-compatible VMs, or Drone-based runners. Each infrastructure option is explained in detail here: [Which Build Infrastructure is Right for Me?](#)

A key architectural difference is the [Harness Delegate](#). The delegate is a lightweight, service-based component you install in your infrastructure (such as a K8s cluster or VM). Think of it like a Jenkins agent, plus credentials manager, plus job orchestrator. It securely connects your environment to the Harness platform, handles pipeline execution, connects to external services (like Docker registries, Git providers, or cloud infra), and ensures operations stay inside your network.

Harness Delegates & Connectors



For teams using Harness Cloud build infrastructure that still need access to internal resources, Harness provides Secure Connect. Secure Connect creates a secure tunnel that allows cloud-hosted builds to access private assets, such as on-premises code repositories or internal artifact registries. It removes the need to allowlist IP addresses and offers a faster, more secure way to bridge cloud and on-prem environments.

While Jenkins users are responsible for managing agents and provisioning access to secrets or cloud APIs through plugins, Harness simplifies these tasks through the Delegate model and features like Secure Connect. This results in less manual setup, improved security, and reduced maintenance effort.

Access Control and Governance

In Jenkins, access control is typically handled through plugins like Role Strategy or Matrix Authorization Strategy. While these provide basic RBAC, configurations are often manual and vary across controllers. Enforcing standards or approvals usually relies on job-level scripting or organizational norms, which can be hard to scale or audit consistently.

Harness includes fine-grained RBAC built into the platform, allowing teams to scope permissions by project, environment, service, or pipeline. Roles can be centrally managed and tied into identity providers via SSO for consistent access control across teams and environments.

Beyond traditional RBAC, Harness supports Policy as Code using Open Policy Agent (OPA). Policies can be defined and stored directly in the Harness platform and applied to specific entities (such as pipelines or stages) at defined points in the delivery workflow. This allows teams to enforce security, compliance, and operational standards in a consistent, codified way.

Jenkins users familiar with DSL-based job controls or homegrown compliance scripts will find the concept familiar, but Harness provides a centralized, auditable, and scalable way to define and enforce rules without scattering logic across jobs.

Debugging and Visibility

In Jenkins, logs are tied to job runs and typically displayed as raw console output. Debugging often involves scrolling through long logs, deciphering stack traces, and mentally reconstructing what went wrong. Visualization and structured context are limited unless additional plugins are installed and configured.

Harness takes a different approach. Logs are automatically structured and tied to specific steps and stages, with direct links between failures and the pipeline execution graph. Beyond that, Harness includes AI-assisted failure analysis built into the platform. When a pipeline fails, Harness AI parses the logs, identifies the likely root cause, and highlights the failure reason, saving developers from log-diving and guesswork.

The core idea of reviewing logs to debug failures remains familiar, but Harness dramatically reduces the time it takes to find and fix issues by combining structured views with intelligent insights out of the box.

Rollbacks, Approvals, and Audit Trails

Rollback strategies in Jenkins are typically custom scripts embedded into jobs. In Harness, rollbacks are treated as first-class citizens with built-in deployment strategies, approval steps, and AI-powered verifications. Harness also includes an audit trail across pipeline edits, secrets, users, and deployments.

Teams used to scripting their way around Jenkins will find these capabilities familiar but more streamlined and centrally visible in Harness.

A Structured Approach to Jenkins Migration: Moving to Harness

Preparation and Assessment

The first step in a successful migration is thorough preparation and assessment:

1. Have a running Cloudbees or Jenkins instance
2. Java 8 to 17 (recommended versions are Java 8, Java 11, or Java 17)
3. Some disk space to store Jenkins traces under the Jenkins Home directory
4. Minimum Jenkins version supported: jenkins/jenkins:2.387.3-lts-jdk11

Planning Your Migration Strategy

Day 0

With a clear understanding of your Jenkins environment, develop a strategic migration plan:

5. Determine the first team to migrate, and ensure one engineer is fully dedicated to supporting the effort
6. Harness will install the **Harness's Jenkins Migration Assistant™**, run a pipeline in Jenkins, and validate initial traces
7. Share the disk space/S3 bucket with Jenkins traces (step 3) with Harness

Week 1

Validate fit

8. Collected traces from your Jenkins instance will be analyzed by the Harness team
9. Harness will assess the number of new plugins required and will work with Harness Engineering to develop them

Week 2

Migrate one pipeline and create a project plan

10. Harness will migrate one critical pipeline end-to-end to Harness.
11. Harness will optimize the pipeline in Harness to use Harness differentiated features (CI Intelligence features, CD steps, STO steps)
12. Help the Harness team create a prioritized list of pipelines to migrate based on:
 - Usage: Prioritize the most used pipelines based on traces and your input to drive Harness adoption
 - Similarity: Choose similar pipelines to maximize reusability through templates

Weeks 3–6

13. Achieve at least 30% Harness adoption

Weeks 6+

14. Scale and shut down Jenkins

Example Conversion: Jenkins to Harness

Below is a simplified example of converting a Jenkins pipeline to Harness CI/CD:

Jenkins Pipeline

```
pipeline {  
  agent any  
  stages {  
    stage('Deploy to Dev') {  
      steps {  
        script {  
          withKubeConfig([credentialsId: 'kubeconfig-dev']) {  
            sh 'kubectl apply -f k8s-config-dev.yaml'  
          }  
        }  
      }  
    }  
  }  
}
```

Harness Equivalent:

```
pipeline:
  name: Deploy to Dev
  identifier: Deploy_To_Dev
  stages:
    - stage:
        name: Deploy To Dev
        identifier: Deploy_To_Dev
        type: Deployment
        spec:
          deploymentType: Kubernetes
          service:
            serviceRef: myService
          environment:
            environmentRef: dev
        execution:
          steps:
            - step:
                name: Rollout Deployment
                identifier: rolloutDeployment
                type: K8sRollingDeploy
```

The Benefits of Harness Platform

Operational Efficiency

Migrating to Harness delivers significant operational improvements:

Reduced Maintenance Overhead: Organizations typically see an 80% reduction in pipeline maintenance overhead after migrating from Jenkins to Harness.

Faster Build and Deployment Cycles: Harness can deliver 80% faster builds and deployments through optimized processes and infrastructure utilization.

Decreased Pipeline Failures: The built-in validation and verification capabilities of Harness result in significantly fewer pipeline failures.

Pipeline Consolidation and Standardization

Harness enables dramatic simplification of CI/CD infrastructure:

Template-Based Approach: Organizations can consolidate thousands of Jenkins pipelines into a handful of standardized templates. For example, Ancestry.com reduced its pipeline sprawl by 80:1 after migrating from Jenkins, cutting maintenance costs by 72%.

Governance and Standardization: Harness automatically applies security, compliance, and best practices to all pipelines, ensuring consistency across the organization.

Developer Experience Enhancements

Harness prioritizes the developer experience:

Intuitive Pipeline Studio: Allows developers to experiment quickly without needing to understand the nuances of YAML, making CI/CD more accessible to all team members.

Self-Service Capabilities: Developers can create and modify pipelines using Harness AI™ without relying on specialized DevOps expertise, reducing bottlenecks.

Rapid Feedback Loops: Lightning-fast builds with test and cache intelligence provide developers with immediate feedback.

Advanced Features Beyond Jenkins

Harness offers capabilities that go beyond what Jenkins can provide:

CI Intelligence Features: Test Intelligence, which selects tests to run based on code changes, Build Intelligence, which caches build tasks for Gradle, Maven, and Bazel, Docker Layer Caching, which caches unchanged Docker layers, etc.

AI-Powered Verification: Continuous verification connects CD pipelines with observability tools to automatically validate deployments and detect issues early.

Policy-Based Governance: Federated policy-based governance, powered by Open Policy Agent (OPA), allows organizations to define production requirements as code.

Comprehensive Audit Trails: Bank-tested audit capabilities track all platform activities, satisfying compliance requirements without compromising developer experience.

Purpose-Built Continuous Delivery

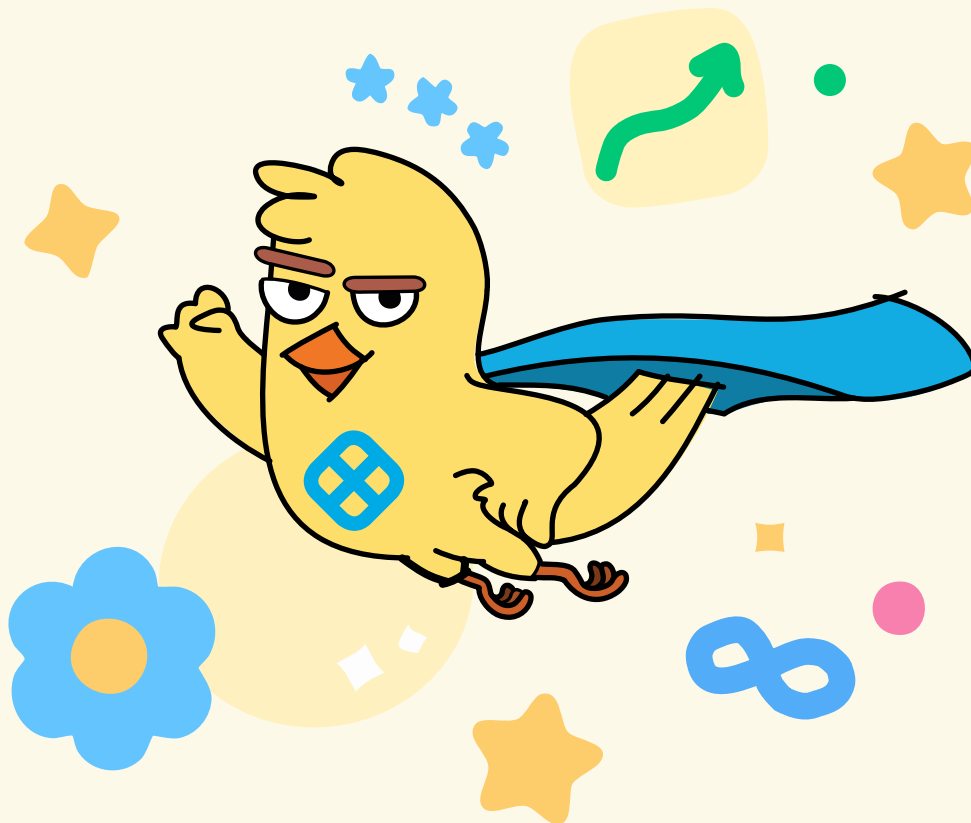
Harness recognizes that CI and CD are distinct disciplines with different requirements:

Specialized CD Platform: Harness was designed from the ground up as a continuous delivery platform, not just an extension of CI capabilities.

Advanced Deployment Strategies: Native support for canary, blue/green, and progressive delivery without complex custom scripting.

Automated Rollbacks: AI-powered rollback capabilities automatically detect and remediate failed deployments.

GitOps Integration: Seamless support for GitOps workflows alongside traditional deployment approaches.



Conclusion

While Jenkins has been a cornerstone of CI/CD for many organizations, its limitations in today's cloud-native, microservices-oriented world are becoming increasingly apparent. The challenges of plugin management, resource consumption, maintenance overhead, and limited support for modern practices are driving organizations to seek alternatives.

Migrating from Jenkins is a substantial undertaking requiring careful planning and execution. However, the benefits of moving to a modern solution like Harness, including reduced maintenance, faster builds, enhanced developer experience, and advanced capabilities, make it a worthwhile investment for organizations looking to stay competitive in today's fast-paced software delivery landscape.

By following a structured approach to migration that includes thorough preparation, strategic planning, incremental implementation, and continuous optimization, organizations can successfully transition from Jenkins to Harness while minimizing disruption and maximizing the return on their investment.

The future of CI/CD lies in AI-powered automation, developer self-service, and built-in governance: all areas where Harness excels. As software delivery continues to evolve, the organizations that embrace these modern approaches will be best positioned to deliver high-quality software at the speed the market demands.

