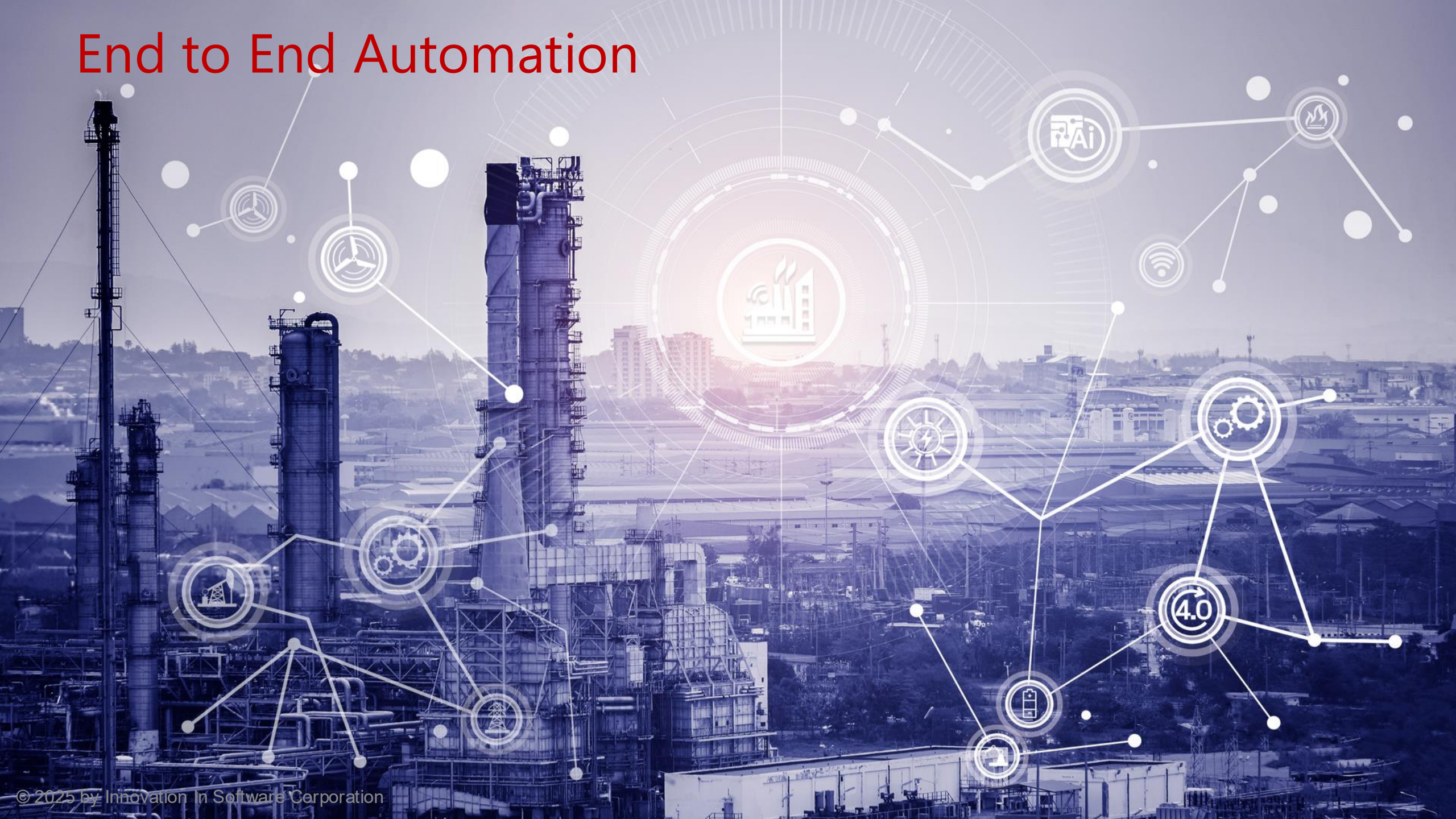
End to End Automation



TEKsystems Global Services

WORKFORCE DEVELOPMENT

Content Usage Parameters

Content refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to copyright protection

2

Content may only be leveraged by students enrolled in the training program

3

Students agree not to reproduce, make derivative works of, distribute, publicly perform and publicly display in any form or medium outside of the training program

4

Content is intended as reference material only to supplement the instructor-led training

WHAT ARE BUILD PIPELINES

Build pipelines are automated workflows that manage the steps required to build, test, and prepare software for deployment. They provide a structured way to ensure that code changes are validated and production-ready at every stage of the development lifecycle.

A build pipeline automates activities like compiling code, resolving dependencies, and generating deployable artifacts, ensuring repeatability and consistency.

- **Stages of a Pipeline:** Typical stages include build (source code compilation), test (validation of code quality), and deploy (packaging and preparing for release).
- **Benefits:** Build pipelines reduce manual errors, accelerate feedback loops for developers, and ensure the software adheres to quality standards before deployment.

KEY COMPONENTS OF A BUILD PIPELINE

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Build application
        run: make build

      - name: Run tests
        run: make test
```

Build pipelines rely on several critical components that work together to automate software delivery effectively. These components are designed to handle various stages of development and ensure the pipeline is efficient and reliable.

- **Source Control Integration:** Pipelines monitor repositories for code changes and trigger workflows automatically upon commits or pull requests.
- **Build Stage:** This stage compiles the source code, resolves dependencies, and generates binaries or other deployable artifacts.
- **Testing Stage:** Automated tests—such as unit, integration, or end-to-end tests—are run to ensure the code behaves as expected and remains stable.
- **Artifact Generation:** Pipelines create deployable outputs, such as container images or packaged files, which are ready for deployment.

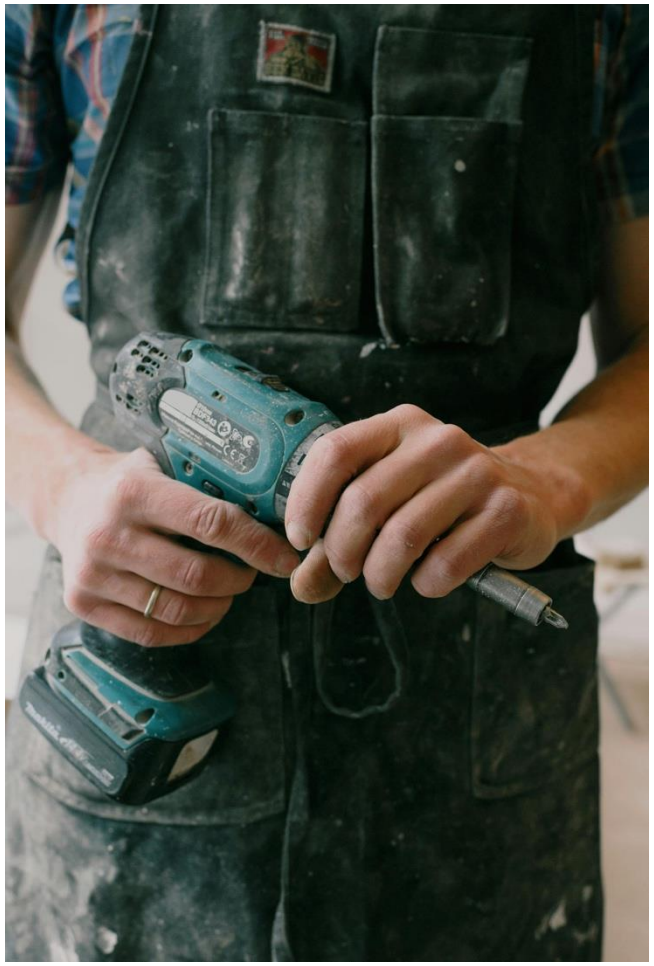
BEST PRACTICES FOR BUILD PIPELINES

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Build application
        run: make build

  # Parallel test jobs to run tests
  # on multiple Node.js versions
  test:
    runs-on: ubuntu-latest
    needs: build  # Ensures the 'test' job runs
                  # only after the 'build' job
    strategy:
      matrix:
        node: [14, 16, 18]  # Creates a matrix to run
                            # tests on Node.js versions
                            # 14, 16, and 18 concurrently
    steps:
      - name: Run tests on Node.js ${{ matrix.node }}
        run: npm test
```

To make build pipelines efficient, maintainable, and reliable, it's essential to follow industry best practices. These practices optimize workflows, reduce errors, and ensure smooth operation across stages:

- **Use Modular Stages:** Divide pipelines into separate stages (build, test, deploy) to isolate issues and make debugging easier.
- **Fail Fast:** Configure pipelines to stop execution immediately upon encountering an error to save time and resources.
- **Parallel Execution:** Optimize performance by running independent tasks, such as testing across environments, in parallel.
- **Monitor and Optimize:** Use logs and analytics to identify bottlenecks, optimize performance, and ensure pipelines remain efficient.
- **Pipeline as Code:** Store pipeline configurations as code in the repository, enabling version control and collaboration.

WHAT ARE BUILD TRIGGERS?



Build triggers are events or conditions that automatically initiate the execution of a pipeline. They eliminate the need for manual intervention and ensure that workflows are executed in response to specific actions or schedules, enabling seamless automation.

Definition: Build triggers define when and how a pipeline starts, based on events like code changes, pull requests, or timed schedules.

Types of Triggers: Triggers can be event-driven (e.g., on a commit or pull request) or time-based (e.g., scheduled builds).

Benefits: Build triggers ensure builds are consistent, reduce manual steps, and allow teams to catch issues early by automating workflow execution.

COMMON BUILD TRIGGER TYPES

yaml

```yaml
name: Build and Test Pipeline

# Define triggers for the workflow
on:
  # Push events: Trigger when changes are pushed to the main branch or tags
  push:
    branches:
      - main
    tags:
      - v*

  # Pull request events: Trigger when a PR is opened or updated, targeting the main branch
  pull_request:
    branches:
      - main

  # Scheduled triggers: Run every day at 2 AM using cron syntax
  schedule:
    - cron: '0 2 * * *'

  # Manual trigger: Allow users to manually start the workflow through the Actions UI
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target environment (e.g., staging, production)'
        required: true
        default: staging
```

Triggers in GitHub Actions are highly customizable and can adapt to various workflow needs. Key types include:

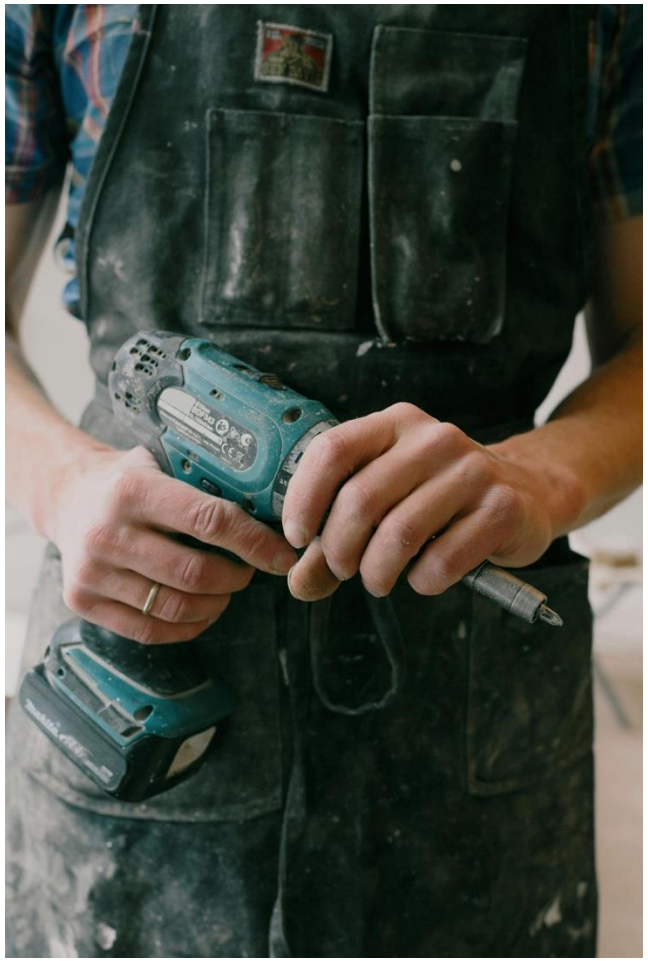
Push Events: Automatically trigger workflows when changes are pushed to a branch or a tag. Useful for continuous integration on the main branch.

Pull Requests: Start workflows when a pull request is opened or updated. Helps validate code changes before merging.

Scheduled Triggers: Use cron syntax to schedule builds at specific intervals, such as nightly tests or weekly deployments.

Manual Triggers: Enable workflows to be triggered manually through the GitHub Actions UI or via API calls for ad-hoc needs.

BEST PRACTICES FOR BUILD TRIGGERS



Configuring build triggers thoughtfully ensures workflows run efficiently and only when necessary. Best practices include:

Scope Triggers to Relevant Changes: Limit triggers to specific branches (e.g., main) or tags to avoid running workflows on irrelevant updates.

Use Filters: Apply paths or paths-ignore to trigger workflows only when specific files or directories are modified.

Combine Triggers: Use a combination of triggers, such as push and pull_request, to address different scenarios like direct commits and code reviews.

Avoid Over-Triggering: Ensure workflows aren’t triggered excessively, which could waste resources or cause unnecessary delays.

WHAT ARE ENVIRONMENT VARIABLES?

```yaml
env:
  NODE_ENV: production

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Print variable
        run: echo $NODE_ENV
```

Environment variables are dynamic values that store configuration data, secrets, or other information required by workflows during execution. They allow developers to customize workflows without hardcoding values directly into the pipeline.
Environment variables are key-value pairs accessible to workflows and steps, used for storing reusable configuration like API keys, URLs, or environment-specific settings.

Scope of Variables: Variables can be scoped globally (workflow-wide), per job, or per step. This flexibility ensures that sensitive or environment-specific data is kept secure and accessible where needed.

Benefits: Environment variables simplify workflows, improve reusability, and make pipelines easier to maintain by centralizing configuration data.

USING ENVIRONMENT VARIABLES IN WORKFLOWS

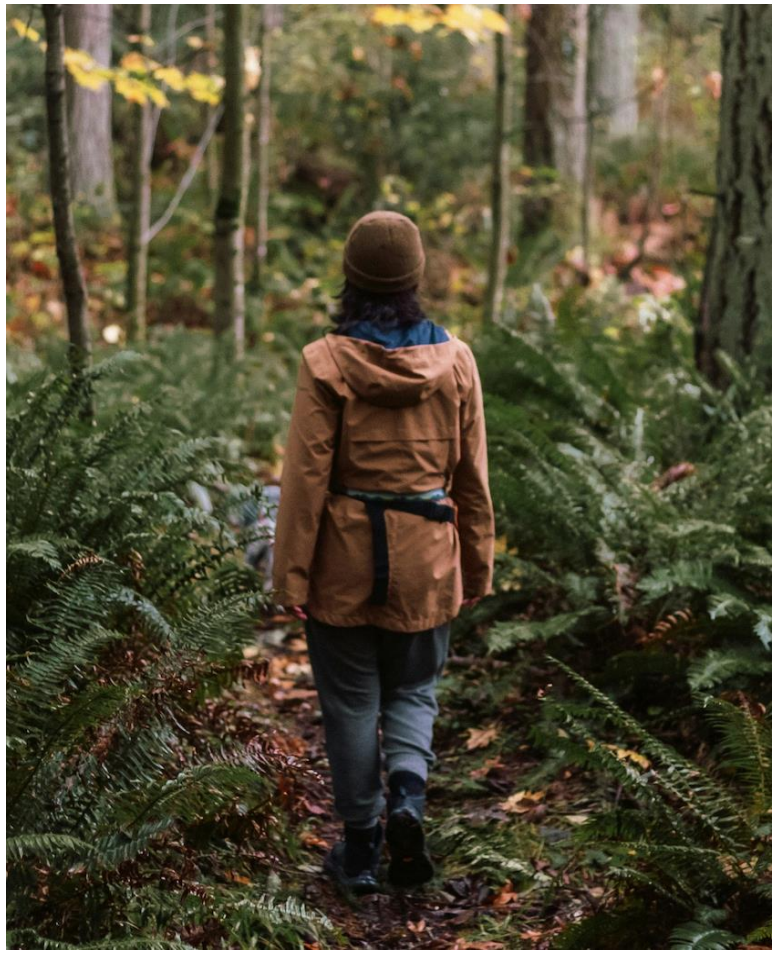
yaml

```yaml
name: Variable Scope Example

on:
  push:
    branches:
      - main

env:  # Workflow-level variables
  WORKFLOW_VAR: "This is accessible to all jobs and steps"

jobs:
  build:
    runs-on: ubuntu-latest

    env:  # Job-level variables
      JOB_VAR: "This is accessible only within the build job"

    steps:
      - name: Print workflow variable
        run: echo "Workflow variable: $WORKFLOW_VAR"

      - name: Print job variable
        run: echo "Job variable: $JOB_VAR"

      - name: Define and print step variable
        run: |
          STEP_VAR="This is only accessible within this step"
          echo "Step variable: $STEP_VAR"
```

Environment variables in GitHub Actions can be set and accessed at various levels, providing flexibility in managing configuration data. Env vars can apply at many different levels:

- **Workflow Level:** Variables defined at this level are accessible throughout the entire workflow.
- **Job Level:** Variables can be scoped to a specific job, ensuring other jobs cannot access them.
- **Step Level:** For maximum isolation, variables can be defined for individual steps.

BEST PRACTICES FOR ENVIRONMENT VARIABLES



Managing environment variables effectively ensures security, maintainability, and clarity in workflows. Key practices include:

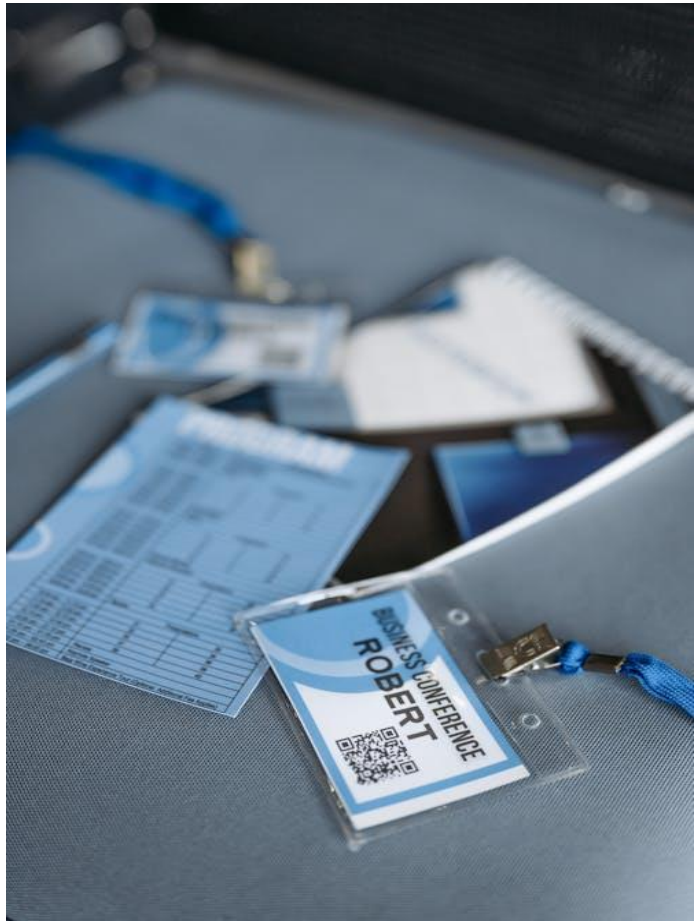
Limit Scope: Define variables only at the level where they are needed (workflow, job, or step) to minimize unintended access.

Avoid Hardcoding: Use variables for values that may change across environments (e.g., development, staging, production) to simplify updates.

Secure Secrets: Store sensitive information like API keys and credentials in GitHub Secrets, which encrypts the data.

Document Variables: Clearly document the purpose and scope of each variable in the repository's README or pipeline documentation.

GITHUB ACTIONS CREDENTIALS



Credentials are sensitive data such as authentication tokens, API keys, or certificates that workflows use to access external services securely. Proper management of credentials is critical for maintaining the security of your workflows and systems.

Credentials refer to any sensitive information required by workflows to interact with external services or perform secure operations (e.g., accessing cloud platforms or private repositories).

- **Storage:** GitHub Secrets is the primary method for securely storing credentials, encrypting the data and restricting access.
- **Usage:** Credentials enable workflows to authenticate securely without exposing sensitive data directly in the pipeline.

MANAGING CREDENTIALS

yaml

```yaml
name: Secure Credentials Example

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the repository code
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Configure AWS Credentials securely using GitHub Secrets
      - name: Configure AWS Credentials
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}  # Injected secret
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}  # Injected secret
        run: echo "AWS credentials configured"

      # Step 3: Deploy application to AWS using injected credentials
      - name: Deploy to AWS
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |
          aws s3 sync ./build s3://my-bucket
```

Credentials are securely managed through GitHub Secrets and are injected into workflows when needed. Key considerations for managing credentials include:

GitHub Secrets: Use the repository's Secrets feature to store credentials securely. Secrets are accessible only to authorized workflows.

Accessing Secrets: Secrets are injected as environment variables and accessed using ${{ secrets.SECRET_NAME }} in workflows.

Scoped Access: Secrets can be scoped at the repository, organization, or environment level to limit access to only the workflows that need them.

BEST PRACTICES FOR CREDENTIALS

Securely managing credentials ensures the safety of your workflows and external services. Follow these practices to maintain security:

Use GitHub Secrets: Always store sensitive information like API keys or tokens in GitHub Secrets instead of hardcoding them in workflows.

Scope Secrets Carefully: Restrict access to secrets based on the principle of least privilege. For example, use environment-level secrets for production credentials.

Rotate Secrets Regularly: Update and rotate secrets periodically to reduce the risk of compromise.

Audit Secret Usage: Regularly review workflows to ensure secrets are being used securely and that no unnecessary secrets are defined.

PARAMETERIZATION OVERVIEW

Parameterization allows workflows and pipelines to use dynamic inputs instead of hardcoding values. By leveraging parameters, pipelines can adapt to different environments, configurations, or inputs, improving flexibility and maintainability.

Parameterization introduces variables that are set dynamically, enabling workflows to behave differently based on the inputs provided.

- **Use Cases:** Deploying to multiple environments (e.g., dev, staging, prod), running tests for different configurations, or customizing resource provisioning.
- **Benefits:** Improves reusability, reduces redundancy, and simplifies updates by centralizing configurable values.

DEFINING AND USING PARAMETERS

```yaml
name: CI/CD Workflow with Parameters

on:
  # Trigger the workflow manually with inputs
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target environment (staging, production)'  # Workflow Input
        required: true
        default: staging

jobs:
  build:
    runs-on: ubuntu-latest

    # Define environment variables for the entire job
    env:
      API_URL: https://api.example.com  # Environment Variable
      DEPLOY_REGION: us-east-1

    steps:
      # Step 1: Checkout repository
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Print workflow input and environment variables
      - name: Print parameters
        run: |
          echo "Target environment: ${{ inputs.environment }}"
          echo "API URL: $API_URL"
          echo "Deploy region: $DEPLOY_REGION"

  test:
    runs-on: ubuntu-latest

    # Use a matrix to run tests across multiple configurations
    strategy:
      matrix:
        node-version: [14, 16, 18]  # Matrix Parameter for Node.js versions
        os: [ubuntu-latest, windows-latest]  # Matrix Parameter for operating systems

    steps:
      # Step 1: Setup Node.js with the current matrix version
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
```

GitHub Actions workflows allow for dynamic behavior by defining and consuming parameters. These parameters control workflow execution and can be defined at different levels or passed from external sources.

Workflow Inputs: Inputs are set at the workflow level, ideal for manual triggers or reusable workflows. For example, users can specify target environments like staging or production when triggering a workflow.

Environment Variables: These store configuration values shared across multiple steps or jobs, such as API URLs or deployment regions, enabling centralized and reusable settings.

Matrix Parameters: Matrices enable running jobs in parallel with varying configurations, such as testing across Node.js versions or operating systems, ensuring broader coverage and efficiency.

BEST PRACTICES FOR PARAMETERIZATION

Effectively parameterizing workflows ensures better reusability, security, and maintainability. Key practices include:

Validate Inputs: Use conditional logic to validate parameter values and prevent invalid configurations.

Keep Parameters Relevant: Only define parameters necessary for the workflow's flexibility and avoid overcomplicating pipelines.

Combine with Secrets: Parameterize sensitive data with GitHub Secrets for security while still enabling flexibility.

Document Parameters: Clearly document what each parameter does and its acceptable values to aid understanding and maintainability.

END-TO-END INFRASTRUCTURE DEPLOYMENT PIPELINE

In this walkthrough, we will build a complete end-to-end infrastructure deployment pipeline using GitHub Actions and Terraform. This pipeline will automate the provisioning of infrastructure and deployment of resources in a secure and reusable manner.

What You Will Learn:

- How to set up a GitHub Actions pipeline for infrastructure deployment.
- Incorporating Terraform to automate provisioning tasks.
- Using parameters, environment variables, and credentials for flexibility and security.
- Testing, validating, and monitoring the pipeline to ensure reliability.

Key Features of the Pipeline:

- Triggered on changes to the repository.
- Fully parameterized for multi-environment support (e.g., staging, production).
- Securely manages sensitive credentials with GitHub Secrets.

CONFIGURE TERRAFORM STEPS

yaml

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.4.0

      - name: Initialize Terraform
        run: terraform init

      - name: Plan Terraform
        run: terraform plan

      - name: Apply Terraform
        run: terraform apply -auto-approve
```

The pipeline should automate Terraform workflows by including three critical steps: initialize, plan, and apply. These steps streamline infrastructure provisioning and ensure consistency.

Initialize: Runs terraform init to set up the working directory, download provider plugins, and configure the backend for state management.

Plan: Executes terraform plan to preview infrastructure changes, showing additions, deletions, or modifications before applying them.

Apply: Runs terraform apply to provision or update resources like virtual machines, storage, or networking, ensuring the infrastructure matches the desired state.

SETTING UP THE PIPELINE

```yaml
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

Creating an end-to-end deployment pipeline starts with setting up the repository and defining the workflow. These foundational steps ensure a structured and reusable approach to infrastructure deployment.

Repository Configuration:

- **Create GitHub Repository:** Set up a repository to store all relevant Terraform configuration files (e.g., main.tf, variables.tf) and the workflow YAML file (.github/workflows/deploy.yml). This ensures that infrastructure code is version-controlled and accessible.

Workflow Triggers:

- **Define Pipeline Triggers:** Configure events to initiate the pipeline. For example, use a push event to automatically start the pipeline when changes are pushed to the main branch.
- **Scope Triggered Events:** Limit triggers to specific branches or tags (e.g., only the main branch) to prevent accidental or irrelevant workflow executions.

ENVIRONMENT SPECIFIC CONFIGURATION

```yaml
inputs:
  environment:
    description: 'Target environment'
    required: true

env:
  AWS_REGION: us-east-1

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Initialize Terraform
        run: terraform init -backend-config="env/${{ inputs.environment }}.tfstate"
```

To make the pipeline reusable and adaptable, incorporate parameters and environment-specific settings. These additions ensure workflows are flexible, handle different environments, and simplify updates without duplicating logic.

Parameterize Target Environments: Use GitHub Actions inputs or env to dynamically specify environments like staging or production. This approach allows the pipeline to adjust its behavior based on the chosen environment, such as stricter validation for production or experimental features for staging.

Environment Variables: Define variables for Terraform settings, such as AWS regions or instance types, to centralize configurations. By storing values like AWS_REGION or INSTANCE_TYPE in environment variables, the pipeline remains consistent across environments, reducing the need for code changes.

MANAGING CREDENTIALS

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.4.0

      - name: Apply Infrastructure
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |
          terraform apply -auto-approve
```

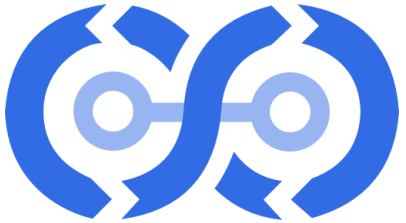
To securely manage credentials in a deployment pipeline, incorporate GitHub Secrets and configure workflows to access them safely. This ensures sensitive data like API keys and cloud provider credentials are protected throughout the deployment process.

Store Secrets: Add credentials such as AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY in the repository's Secrets section to encrypt and restrict access.

Inject Secrets into Workflows: Use secrets within the workflow by mapping them to environment variables, allowing secure access during Terraform execution.

Lab: GitHub Actions

GitOps



GitOps

GitOps uses Git repositories as a single source of truth to deliver infrastructure as code. Submitted code checks the CI process, while the CD process checks and applies requirements for things like security, infrastructure as code, or any other boundaries set for the application framework. All changes to code are tracked, making updates easy while also providing version control should a rollback be needed.

GitOps delivers:

- A standard workflow for application development
- Increased security for setting application requirements upfront
- Improved reliability with visibility and version control through Git
- Consistency across any cluster, any cloud, and any on-premise environment

GitOps

The demand to deliver applications more rapidly and reliably grows across all industries. Organizations increasingly adopt DevOps practices such as continuous integration and continuous delivery (CI/CD) to streamline their software delivery pipelines and improve overall quality.

GitOps workflows enhance this approach by enabling automated, declarative management of application deployments and Kubernetes cluster configurations using tools like Argo CD.

GitOps

By using the same Git-based workflows that developers are familiar with, GitOps expands upon existing processes from application development to deployment, application life cycle management, and infrastructure configuration.

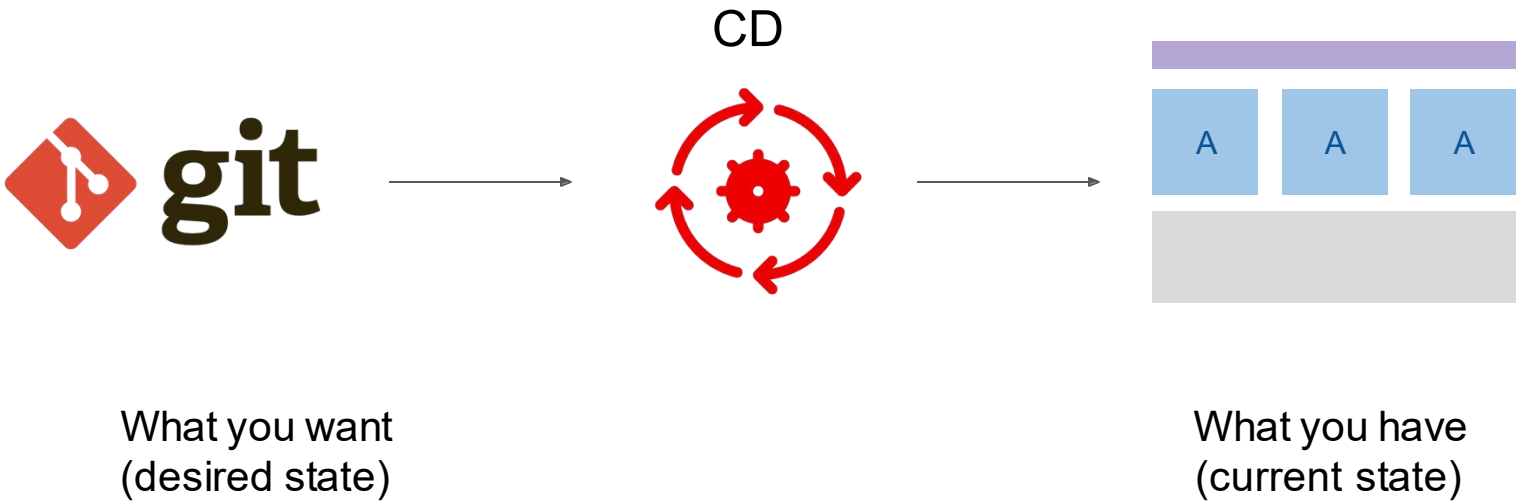
Every change throughout the application life cycle is traced and auditable in the Git repository. Making changes via Git means developers can finally do what they want: code at their own pace without waiting for resources to be assigned or approved by operations teams.

GitOps

For operations teams, having visibility into changes enables them to trace and resolve issues quickly, ultimately improving the overall security posture of the organization. With an up-to-date audit trail, organizations can minimize the risk of unauthorized changes, rectify them before they are implemented, and maintain a secure production environment.

GitOps Workflow

A declarative approach to application delivery

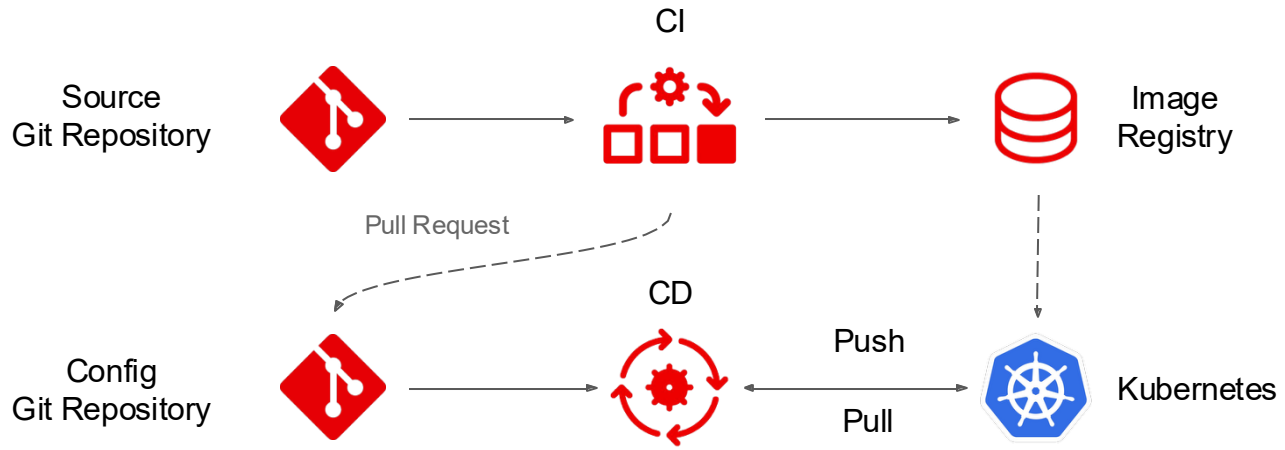


GitOps

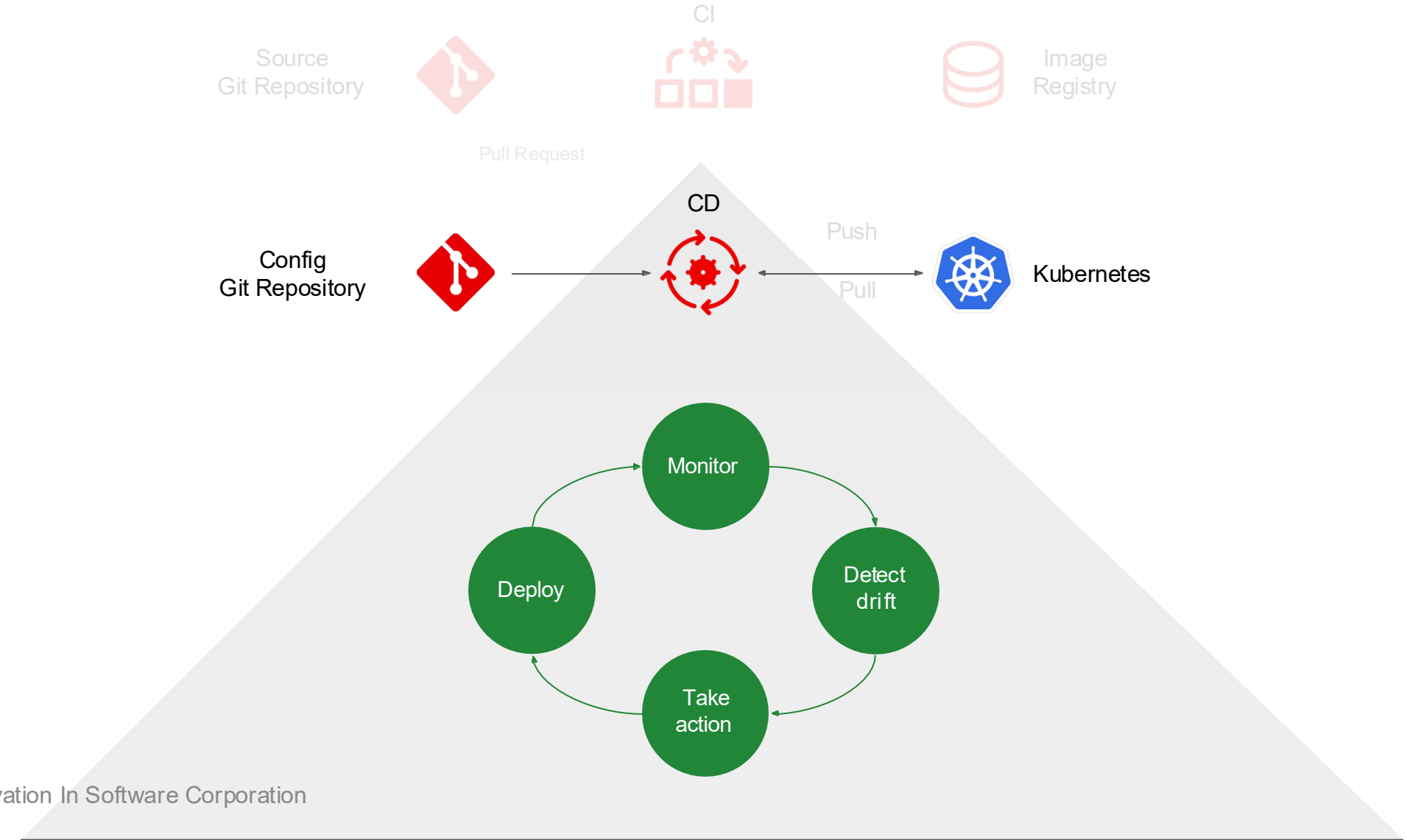
To get started with GitOps, you need declaratively managed infrastructure. Because of this, GitOps is often used as an operating model for Kubernetes and cloud-native application development and can enable continuous deployment for Kubernetes.

However, using Kubernetes is not a requirement of GitOps. GitOps is a technique that can be applied to other infrastructure and deployment pipelines.

The GitOps Application Delivery Model

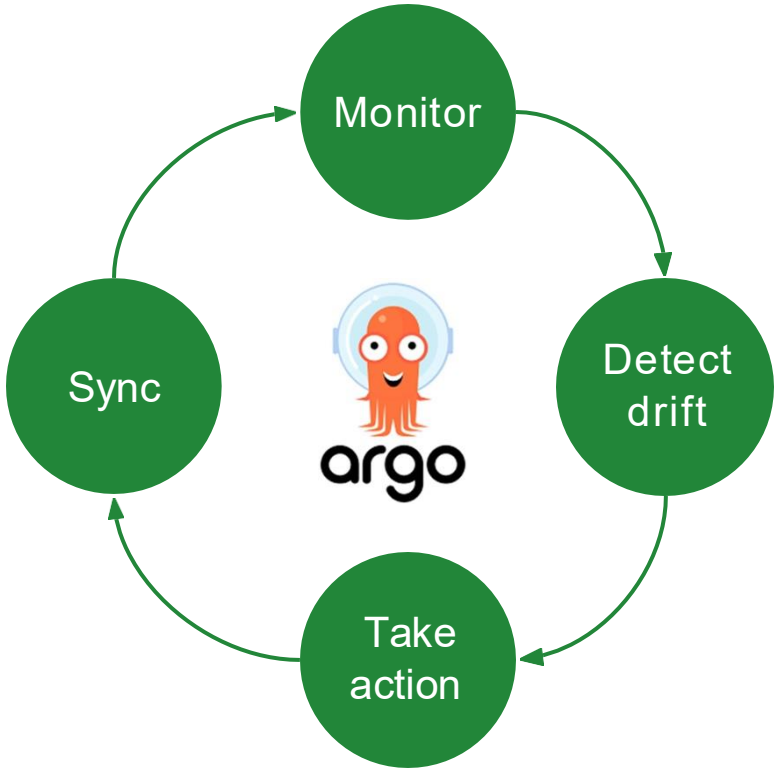


The GitOps Application Delivery Model

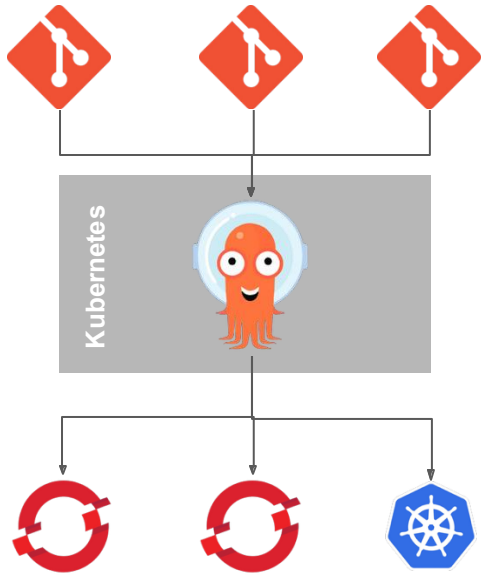


GitOps: Argo CD

- Cluster and application configuration versioned in Git
- Automatically syncs configuration from Git to clusters
- Drift detection, visualization and correction
- Granular control over sync order for complex rollouts
- Rollback and rollforward to any Git commit
- Manifest templating support (Helm, Kustomize, etc)
- Visual insight into sync status and history

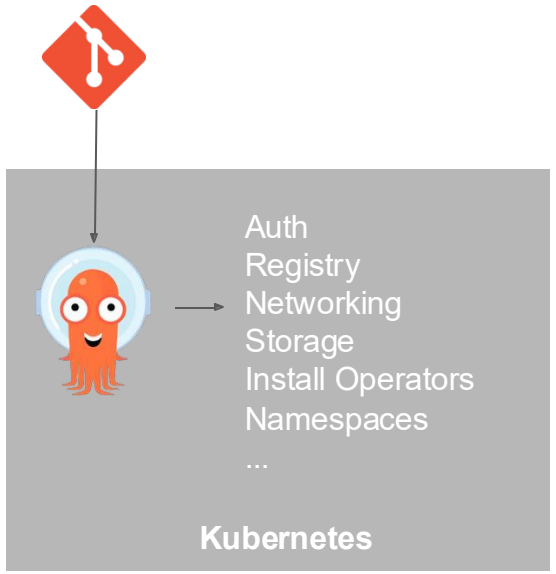


GitOps: Flexible Deployment Strategies



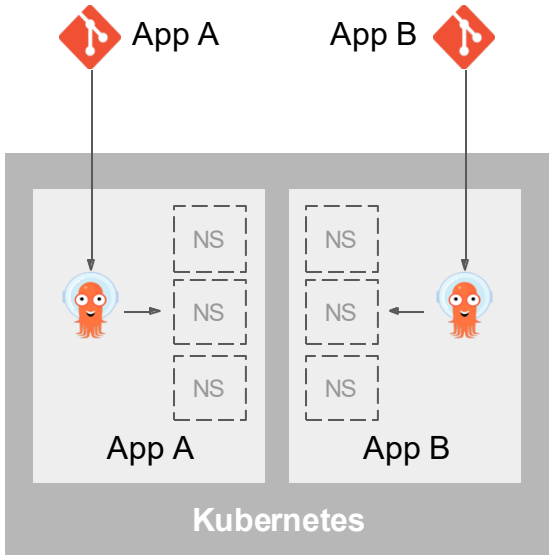
Central Hub (Push)

A central Argo CD pushes Git repository content to remote Kubernetes clusters



Cluster Scoped (Pull)

A cluster-scope Argo CD pulls cluster service configurations into into the cluster.



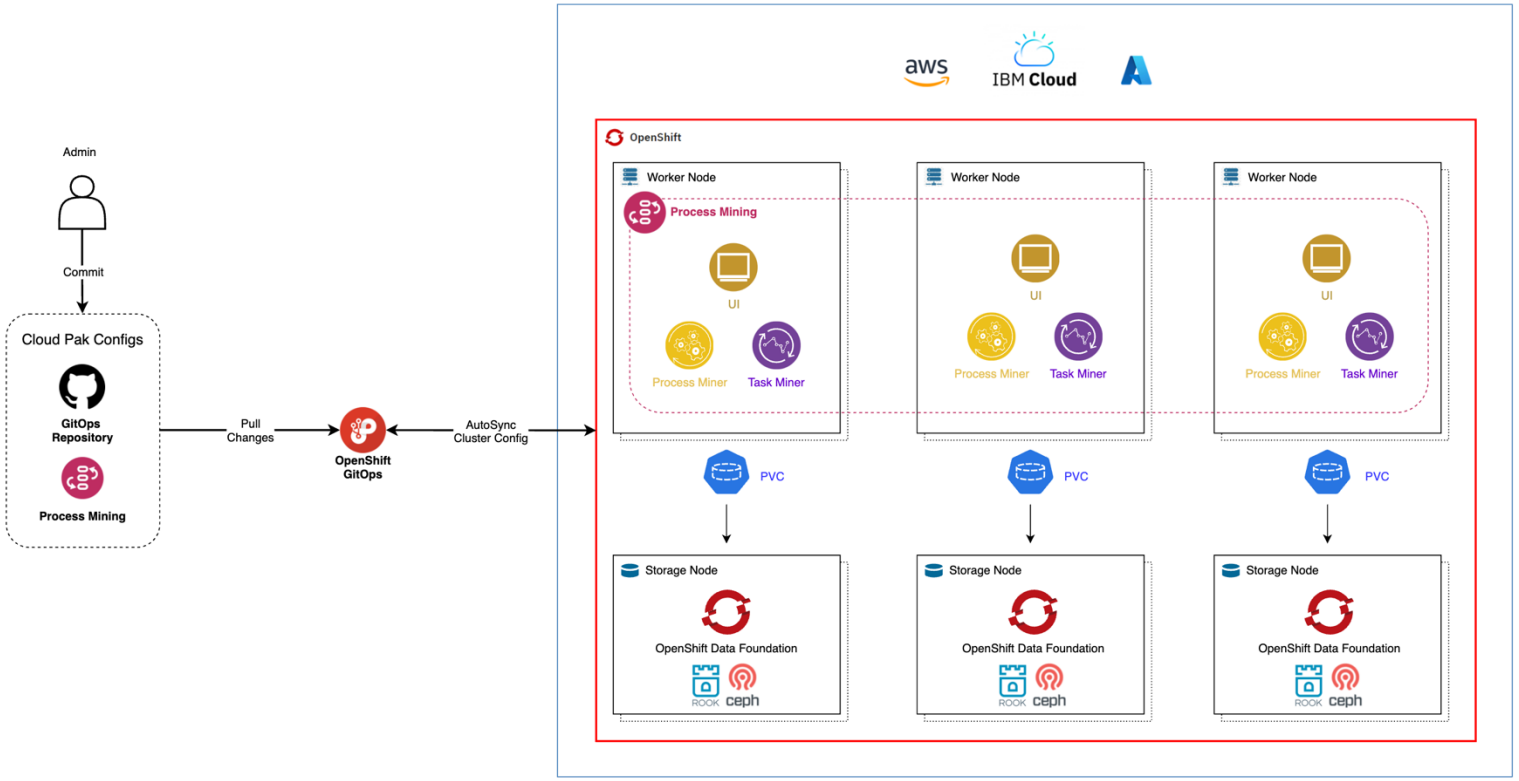
Application Scoped (Pull)

An application scoped Argo CD pulls application deployment and configurations into app namespaces

GitOps

Argo GitOps

- Admin commits changes to the GitOps repository containing configs.
- GitOps pulls changes from the repository.
- GitOps auto-syncs the cluster configuration.
- Process Mining components are deployed and managed.
- Supports multi-cloud environments: AWS, IBM Cloud, Azure.



This diagram illustrates the GitOps workflow for managing and deploying Process Mining applications across a multi-cloud environment.

GitOps

Git has been at the center of software development for a long time, and many teams have adopted the Git pull-request workflow for developing code. GitOps is an approach to continuous delivery (CD) that treats Git as the single source of truth for all configurations, including infrastructure, platform, and application settings. Teams can then leverage Git workflows to streamline cluster operations and application delivery, enabling predictable, more secure, and repeatable changes to clusters.

GitOps

Another benefit of GitOps is that it enhances observability and visibility of the actual state, allowing possible configuration drifts to be detected easily and immediately through the GitOps workflow.

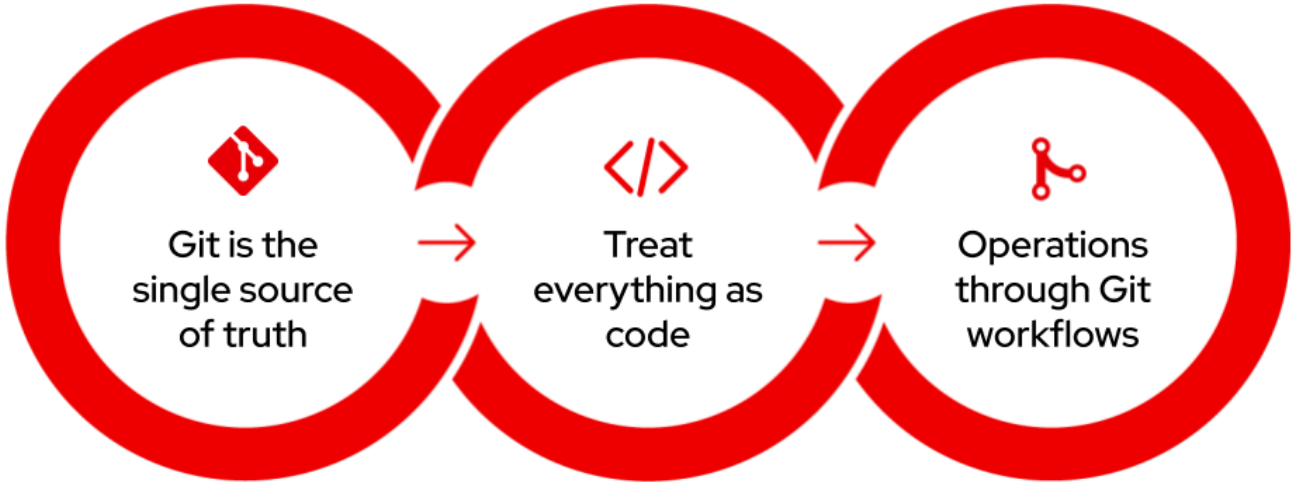
GitOps enables complete transparency through Git audit capabilities and provides a straightforward mechanism for rolling back to any desired version across multiple Kubernetes clusters.

GitOps

GitOps centralizes configuration and infrastructure management in a Git repository. This ensures all changes are version-controlled and auditable, enhancing transparency and collaboration.

By treating infrastructure, configurations, and applications as code, GitOps ensures consistent and repeatable deployments. This allows for automated testing, versioning, and rollback, reducing errors and improving reliability.

GitOps uses Git workflows to automate operational tasks. This includes pull requests for changes, peer reviews, and controlled, predictable deployments, enhancing efficiency.



Discussion:

What is GitOps?

Discussion:

What is GitOps?
Git is the source of truth declarative infrastructure and applications.

Discussion:

What are some benefits of GitOps?

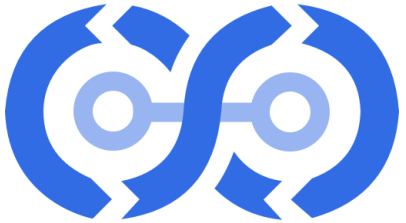
Discussion:

What are some benefits of GitOps?

- A standard workflow for application development
- Improved reliability with visibility and version control through Git
- Consistency across any cluster, any cloud, and any on-premise environment


Questions

Argo CD



Argo CD

Argo CD is a declarative continuous delivery tool for Kubernetes. It can be used as a standalone tool or as part of your CI/CD workflow to deliver the necessary resources to your clusters.

In order to manage infrastructure and application configurations aligned with GitOps, your Git repository must be the single source of truth. The desired state of your system should be versioned, expressed declaratively, and pulled automatically. This is where Argo CD comes in.

Argo CD

Argo CD is a tool for globally deploying custom resources from a Git repository to your Kubernetes clusters, serving as the primary source of truth.

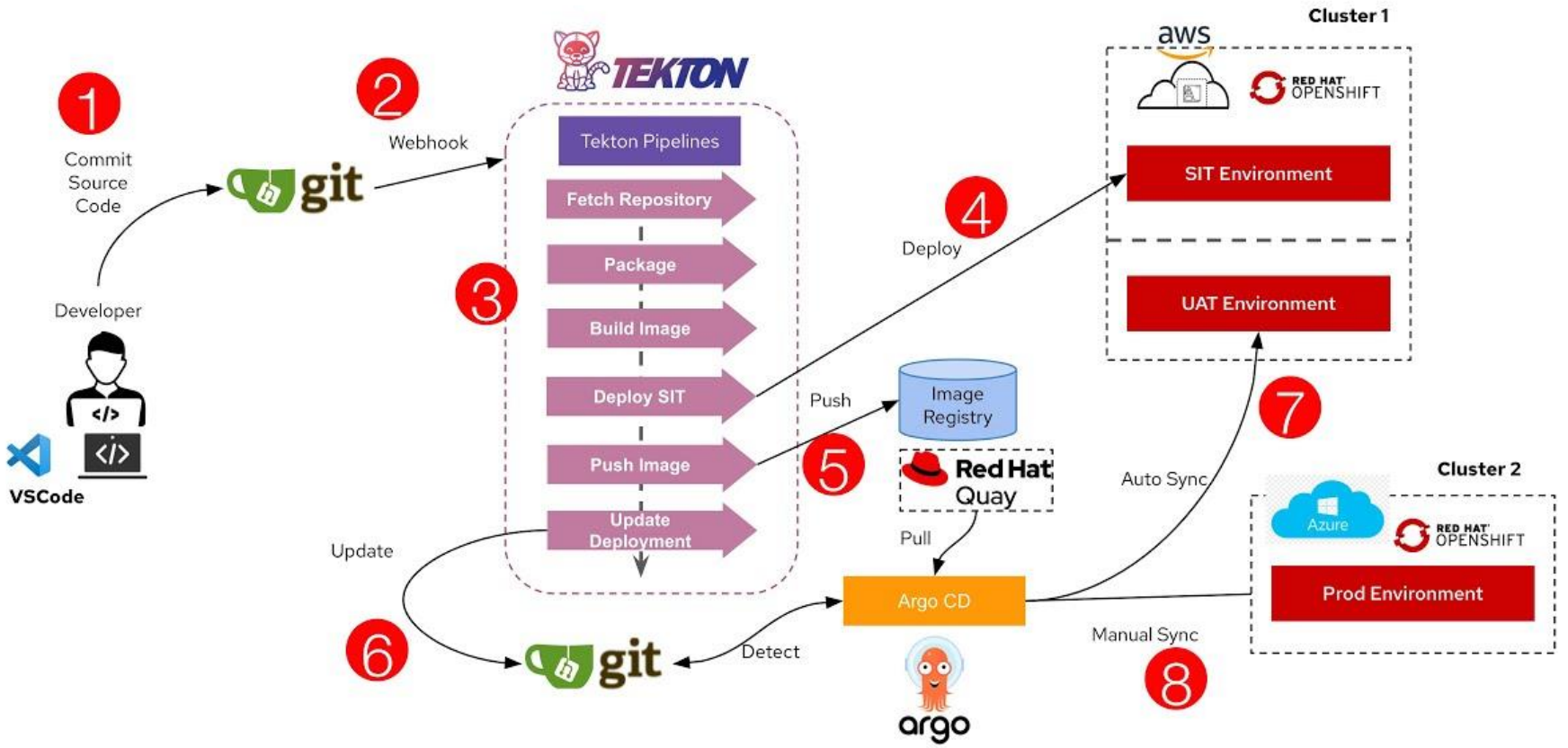
These resources can encompass application definitions, configurations, and target states for your environments. With Argo CD, all these items can be version-controlled, providing a comprehensive solution for managing deployments in a streamlined and efficient manner.

Argo CD

Using Argo CD to continuously deliver these resources can ensure that your applications remain in sync with your desired states.

Argo CD acts as the application controller, continuously checking the Git repository and the applications for parameters defined by the cluster admin.

GitOps diagram

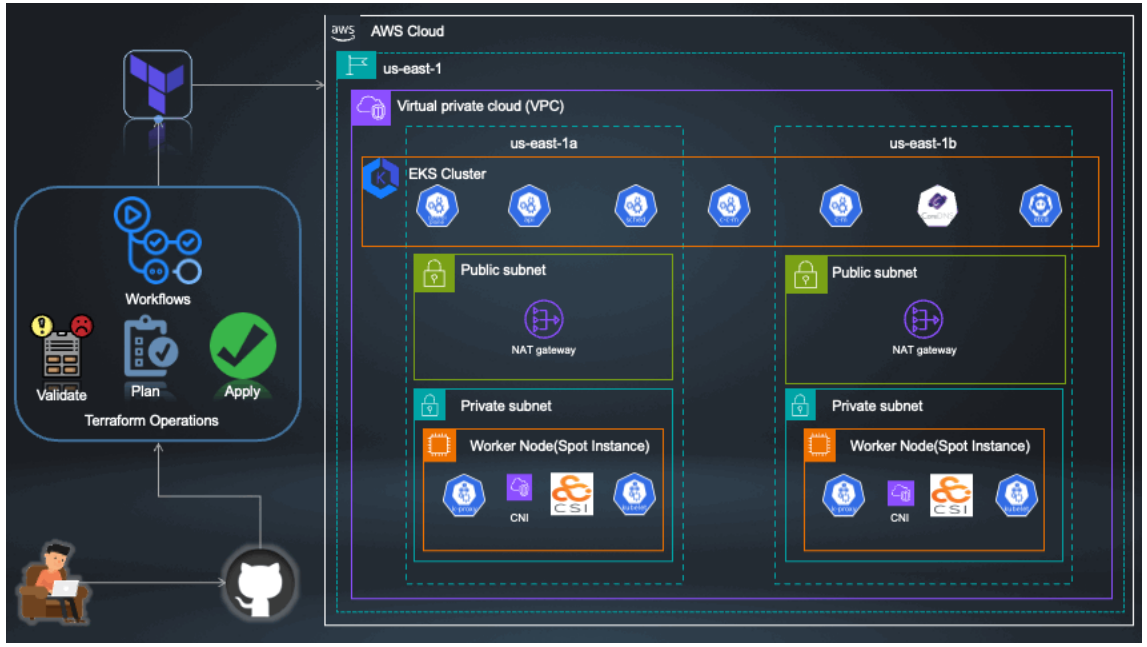



Questions

GitOps workflow

Developer Workflow:

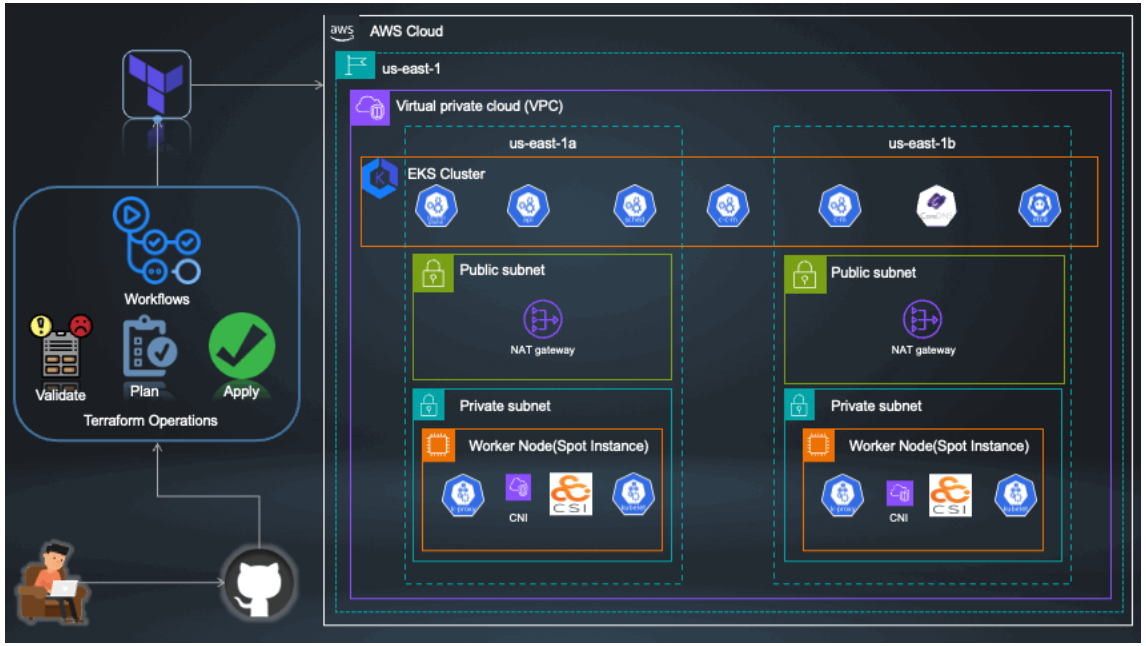
- A developer pushes code or infrastructure changes to GitHub
- This triggers GitHub Actions workflows for automated operations
- CI/CD pipeline handles both infrastructure and Kubernetes application deployments



GitOps workflow

GitHub Actions CI/CD pipeline runs Terraform to:

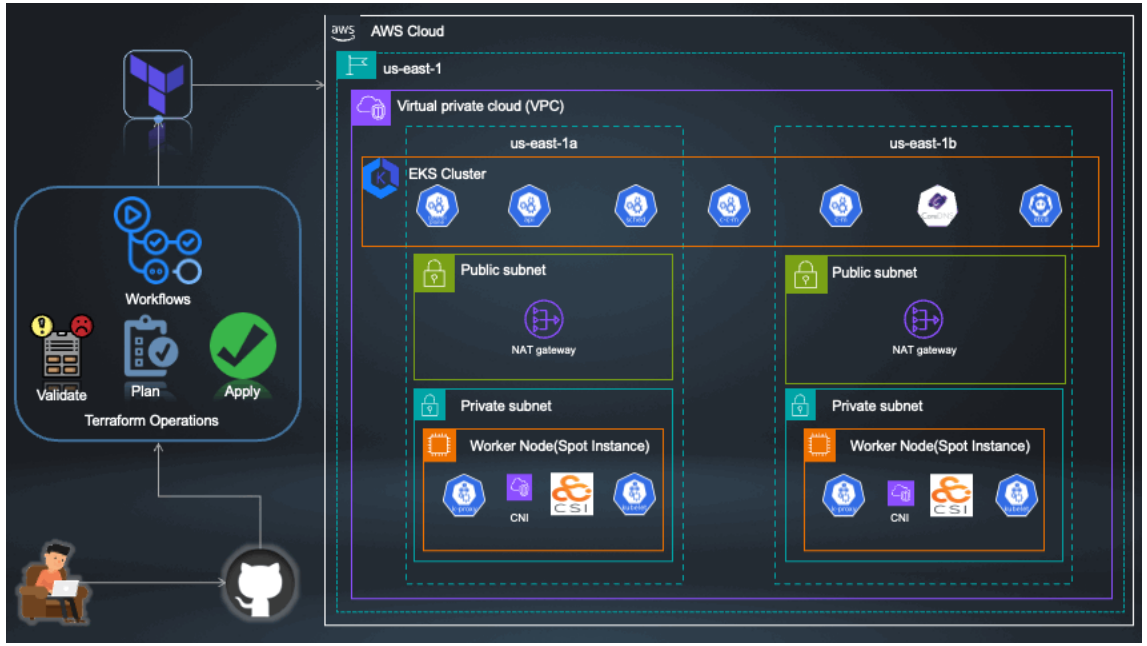
- Provision a Virtual Private Cloud (VPC) in us-east-1
- Deploy an EKS Cluster with managed Kubernetes control plane
- Create Public and Private Subnets across us-east-1a and us-east-1b
- Set up NAT Gateways for controlled internet access
- Launch Worker Nodes (Spot Instances) for cost-efficient scaling



GitOps workflow

CI/CD Pipeline Stages

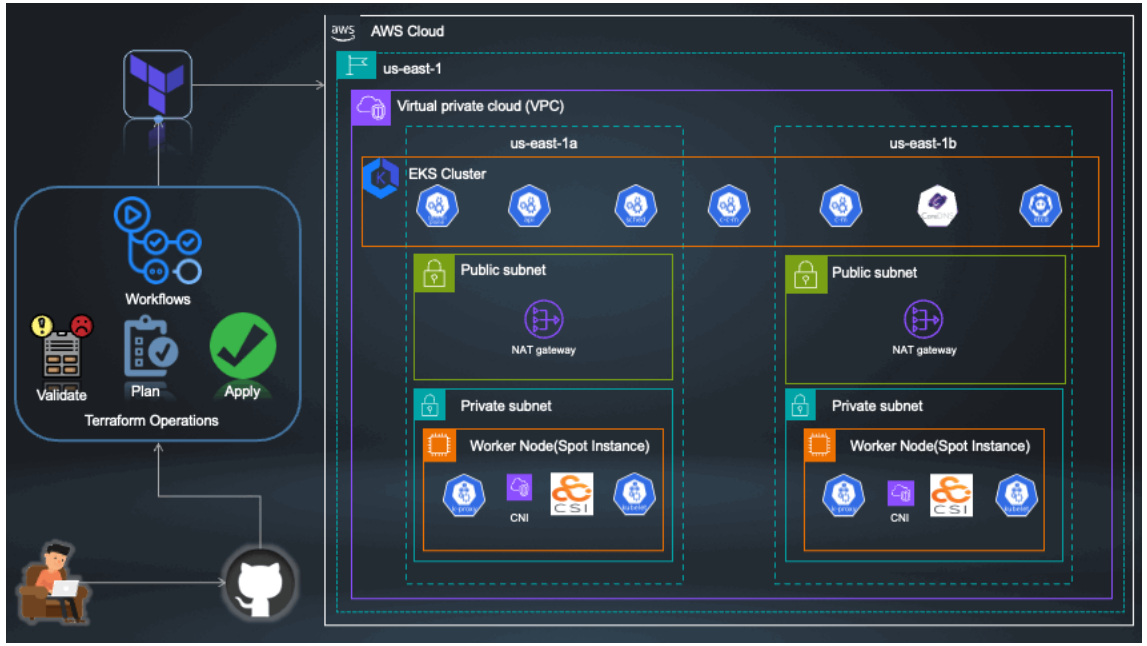
Stage	Action
Validate	Check Terraform syntax and configuration
Plan	Preview infrastructure changes
Apply	Provision or update infrastructure



GitOps workflow

Complete CICD & GitOps workflow

- End-to-End Automation using GitHub Actions
- Infrastructure as Code with Terraform for repeatability
- High Availability across multiple availability zones
- Cost Efficiency with Spot Instances
- Simplified Networking using NAT Gateways and Subnet Design



Lab: GitHub Actions EKS

Lab: GitOps End-to-End Pipeline