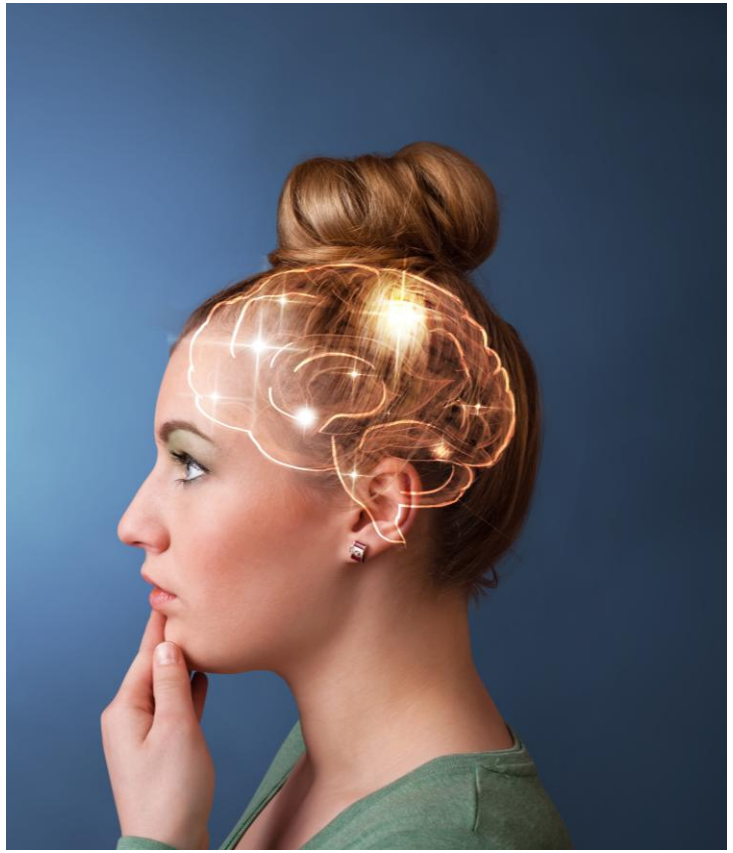
Infrastructure Automation Tools

TEKsystems Global Services

WORKFORCE DEVELOPMENT

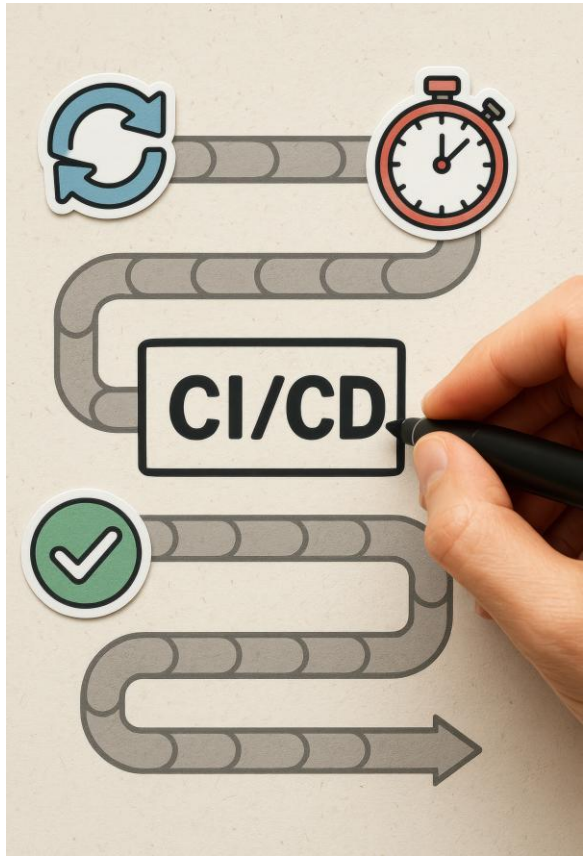
CI/CD and Infra Automation

Course Learning Objectives



- Use GitHub Actions and Jenkins for CI and CD
- Build and use Docker images in automation workflows
- Set up secure cloud authentication with GitHub OIDC
- Deploy resilient AWS infrastructure with Terraform and Packer
- Apply GitOps with Argo CD for releases, rollbacks, and blue or green deployments
- Combine Ansible and Terraform for full stack automation

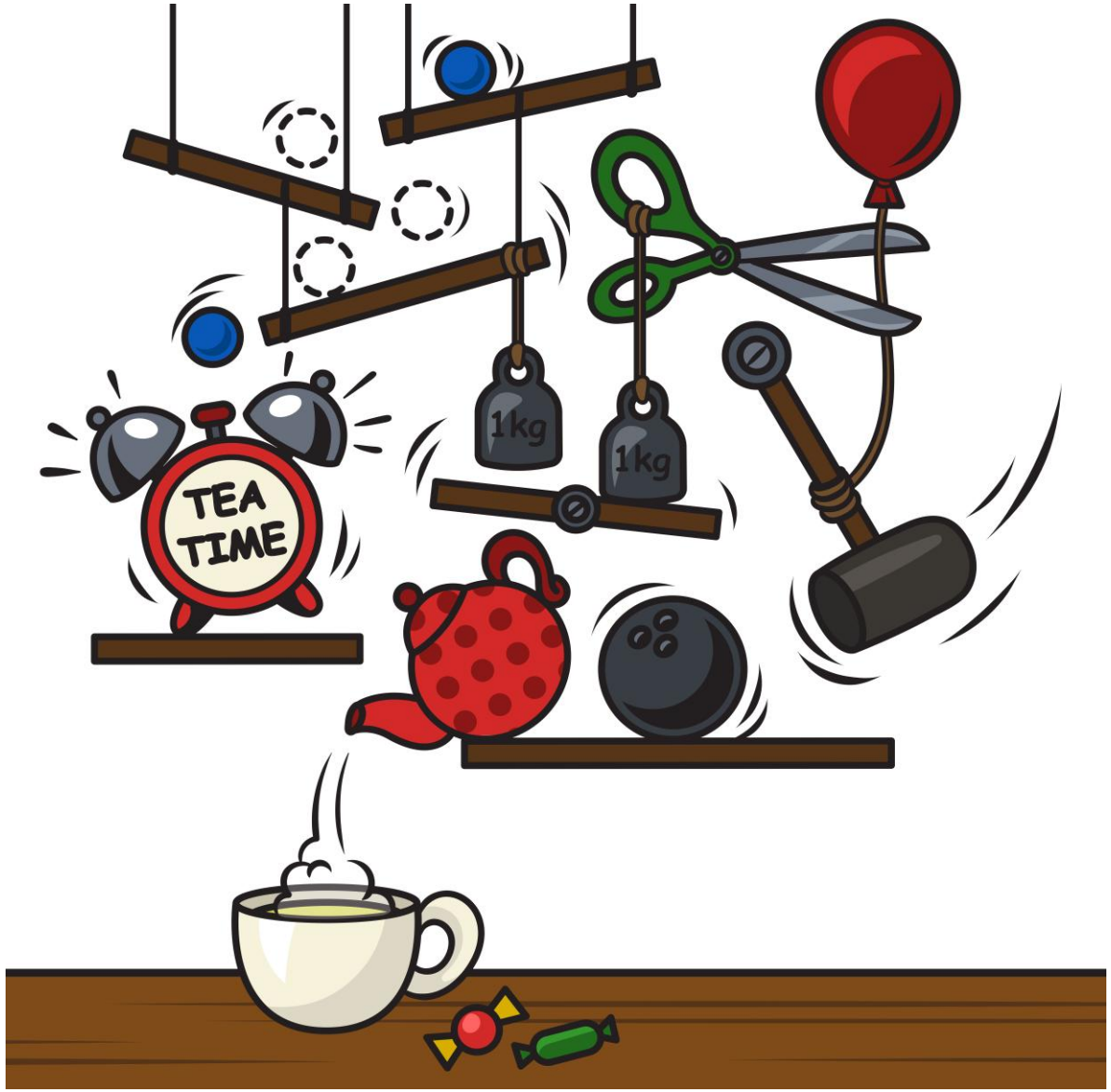
Learning Objectives



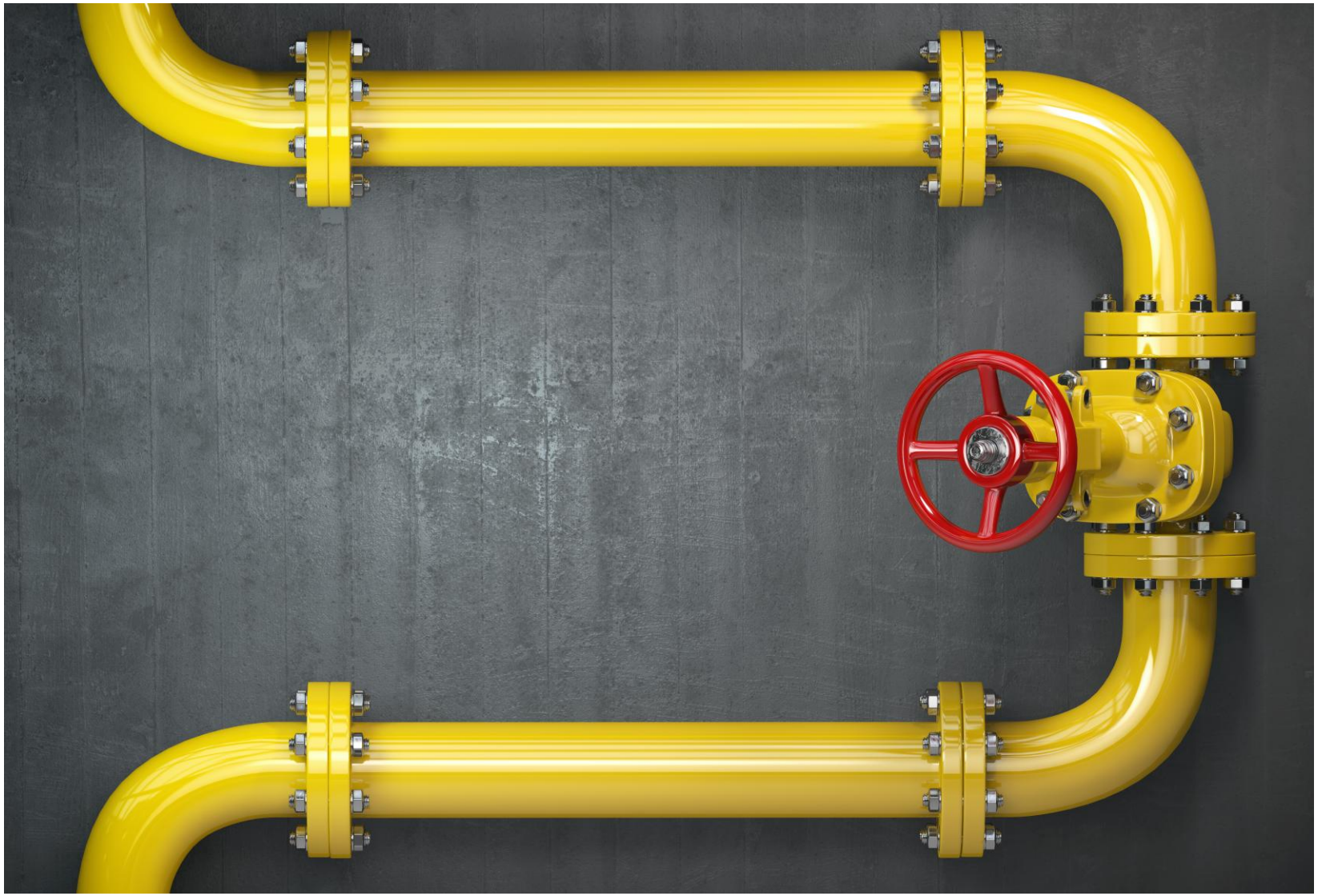
Day 1 Objectives

- Understand the purpose and flow of a CI workflow
- Create your first GitHub Actions pipeline to automate simple tasks
- Build a Docker image and push it to Docker Hub from CI
- Set up GitHub OIDC with AWS to remove long lived credentials
- Understand how pipelines support Terraform, Ansible, and future automation work

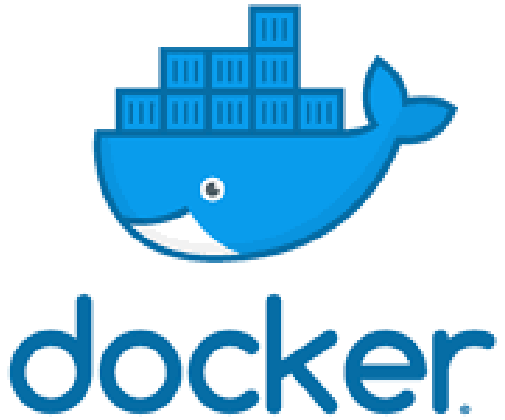
CI CD



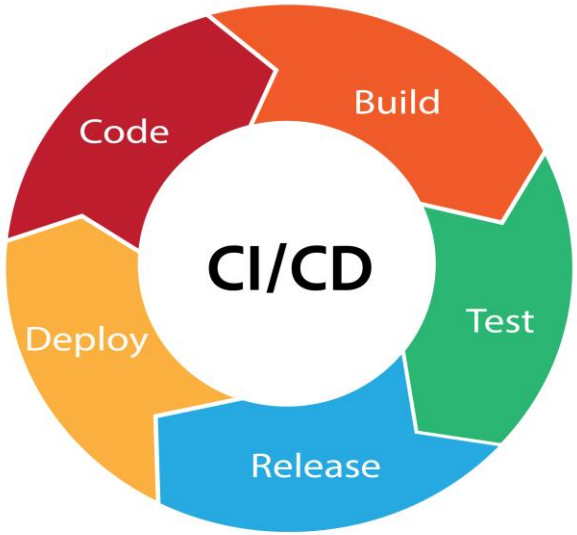
Pipelines



Automation tools



CI/CD



Continuous Integration (CI)
Automatically builds, tests, and validates code every time developers push changes.

Continuous Delivery (CD)
Keeps the application always ready for release with automated packaging and staging.

Continuous Deployment (CD)
Automatically deploys every approved change directly to production or staging envs.

Pipelines

```yaml
name: Multi-Job Workflow

on:
  workflow_dispatch:

jobs:
  job1:
    name: First Job
    runs-on: ubuntu-latest

    steps:
      - name: Step 1
        run: echo "This is Job 1, Step 1"

      - name: Step 2
        run: echo "This is Job 1, Step 2"

  job2:
    name: Second Job
    runs-on: ubuntu-latest
    needs: job1  # Waits for job1 to complete

    steps:
      - name: Step 1
        run: echo "This is Job 2, Step 1"

      - name: Step 2
        run: echo "This is Job 2, Step 2"
```

A pipeline is an automated sequence of steps that takes code from commit to deployment.

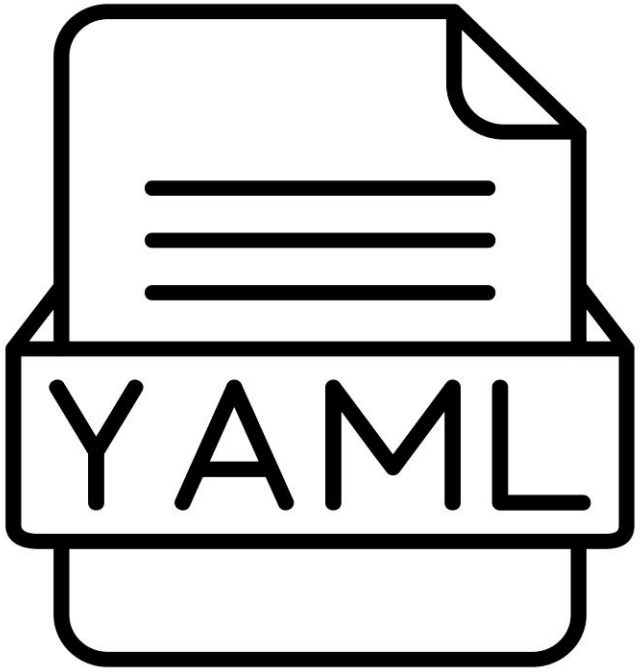
Workflow — The entire automated process

Jobs — Units of work that run tasks in the pipeline

Steps or Stages — Individual actions inside a job

Triggers — Events that start the workflow, such as a push or pull request

YAML



Stands for "Yet Another Markup Language". It Also stands for "YAML Ain't Markup Language."

- Many platforms use YAML to define automation, including CI and CD pipelines.
- Uses indentation to show structure
- Data is represented as keys and values
- Nested blocks define hierarchy

Ephemeral

Pipelines run on **ephemeral servers** provided by the CI platform.

Key points

- Each workflow starts on a fresh virtual machine
- Usually a Linux runner unless otherwise specified
- Files, state, and variables **do not persist** between runs
- Ensures clean, isolated, reproducible builds every time

When do they run?

Workflows can start automatically or by user action.

Manual — Started on demand

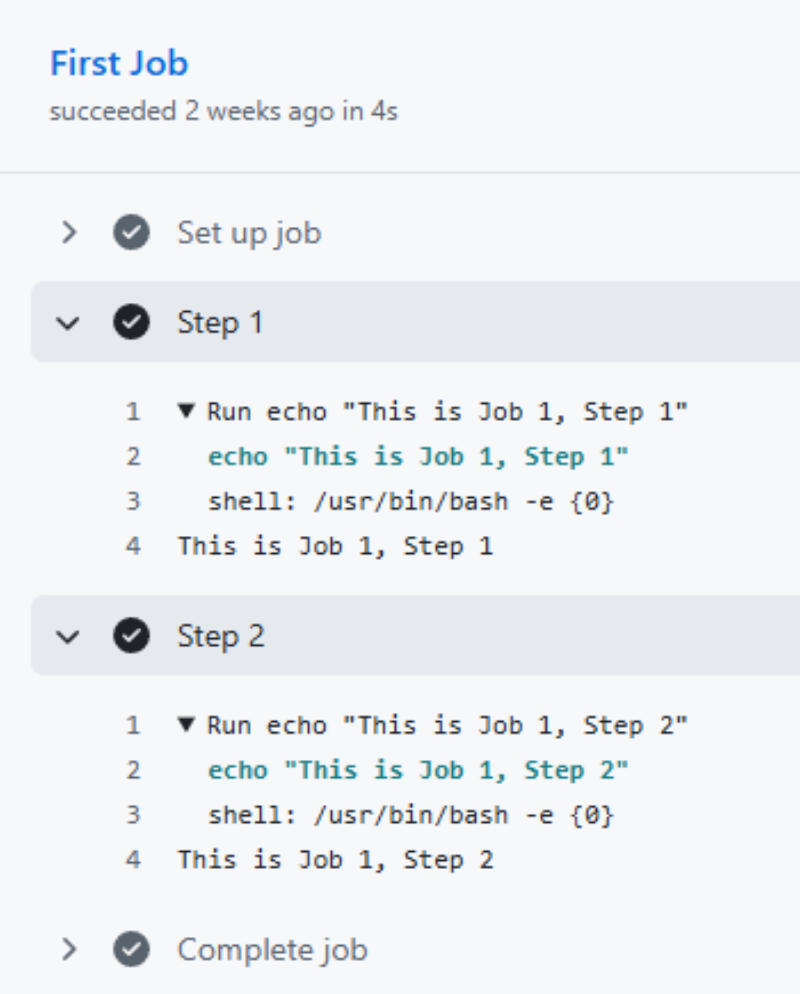
On Push — Runs when code is pushed to the repository

Pull Request — Runs when a PR is opened or updated

Scheduled — Runs on a timer such as daily or hourly

Conditional — Runs only when rules or filters match

What is a Job?



A job is a collection of steps that run in order on a runner.

A job groups related actions in a workflow

Each step runs a specific task

Steps can execute shell commands on the underlying OS

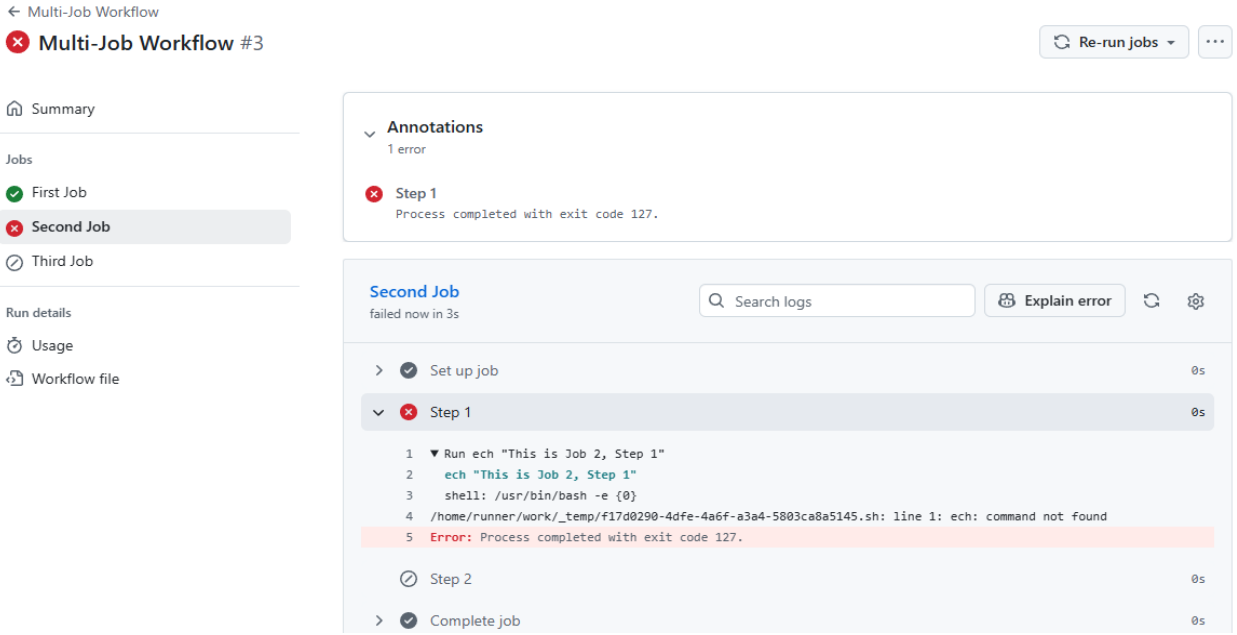
Revisiting the Pipeline

```yaml
name: Multi-Job Workflow

on:
  workflow_dispatch:

jobs:
  job1:
    name: First Job
    runs-on: ubuntu-latest

    steps:
      - name: Step 1
        run: echo "This is Job 1, Step 1"

      - name: Step 2
        run: echo "This is Job 1, Step 2"

  job2:
    name: Second Job
    runs-on: ubuntu-latest
    needs: job1  # Waits for job1 to complete

    steps:
      - name: Step 1
        run: echo "This is Job 2, Step 1"

      - name: Step 2
        run: echo "This is Job 2, Step 2"
```

Consider this pipeline again. Identify the job and its steps, and notice how each step runs in order.

This sequence forms the workflow that moves code through the pipeline.

Step Failures



If a step fails, the rest of the job stops immediately.

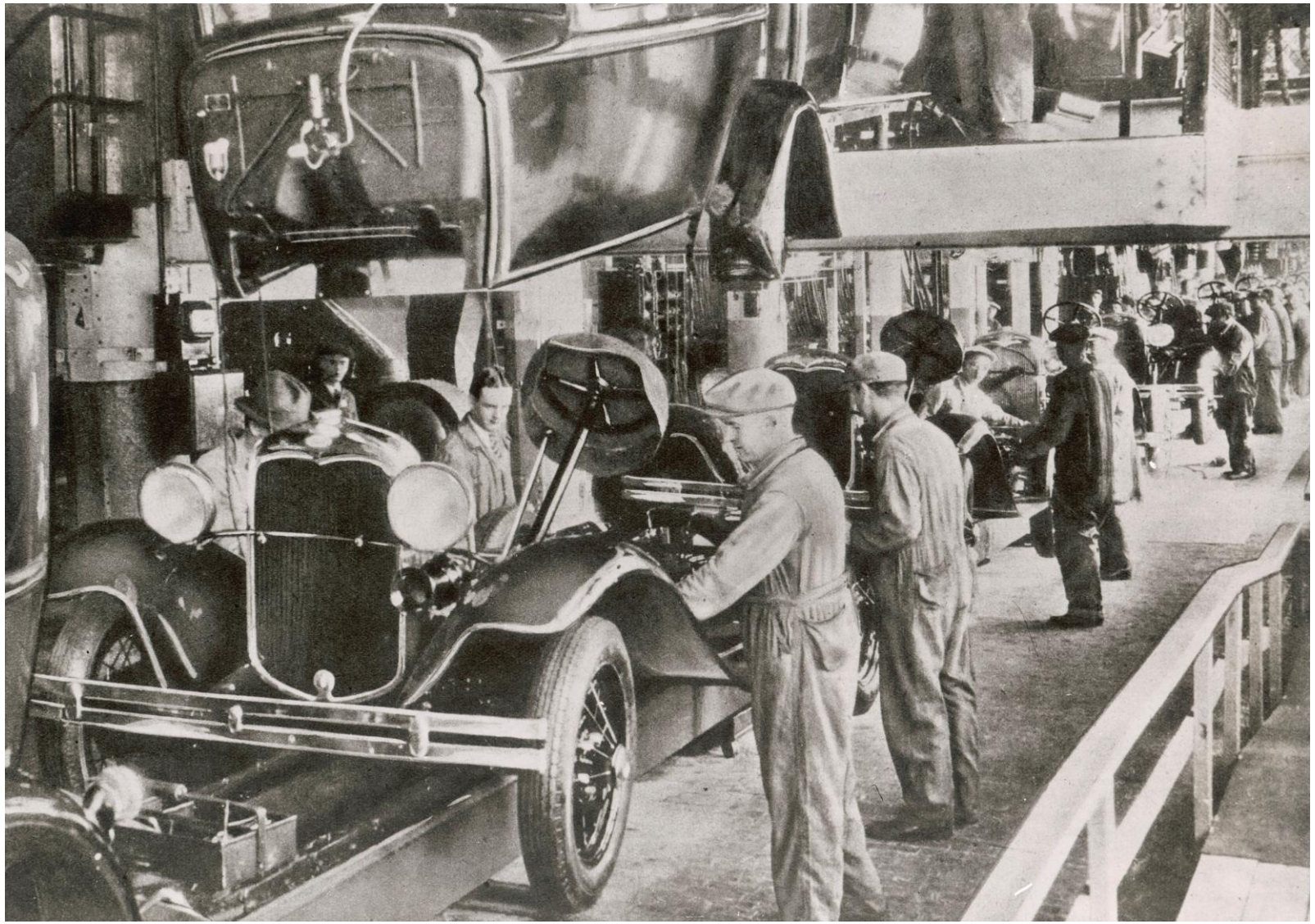
A failed step stops all later steps in that job When a job fails, dependent jobs do not run. This protects the pipeline from deploying broken code.

Nuances of GitHub Workflows

Important workflow behaviors

- Variables created inside a step do not persist to the next step unless exported
- Secrets can be accessed securely through the secrets context
- Inputs and parameters can be passed between jobs using outputs
- Jobs run in isolation unless dependencies are defined
- Environment variables can be set at the workflow, job, or step level

CI CD



POP QUIZ:

What is a common trigger for a GitHub Actions workflow?

A. When a Docker image is pushed
B. When an EC2 instance launches
C. When code is pushed to a specific branch in a repository
D. When Terraform applies changes

POP QUIZ:

What is a common trigger for a GitHub Actions workflow?

A. When a Docker image is pushed
B. When an EC2 instance launches
C. **When code is pushed to a specific branch in a repository**
D. When Terraform applies changes

POP QUIZ:

What does the run: keyword do in a step?

A. Creates a new job
B. Runs the job that it specifies
C. Executes shell commands
D. Defines environment variables

POP QUIZ:

What does the run: keyword do in a step?

A. Creates a new job
B. Runs the job that it specifies
C. **Executes shell commands**
D. Defines environment variables

POP QUIZ:

What happens if a step fails?

A. The workflow retries automatically
B. All future steps in the job stop running
C. The job continues normally
D. The repository is locked

POP QUIZ:

What happens if a step fails?

A. The workflow retries automatically
B. **All future steps in the job stop running**
C. The job continues normally
D. The repository is locked

POP QUIZ:

What is a *job* in a GitHub Actions workflow?

A. A complete CI/CD pipeline
B. A group of steps that run on a runner
C. A file in the repository
D. A single shell command

POP QUIZ:

What is a *job* in a GitHub Actions workflow?

A. A complete CI/CD pipeline
B. **A group of steps that run on a runner**
C. A file in the repository
D. A single shell command

POP QUIZ:

When you run the same GitHub Actions workflow multiple times, where does each run execute?

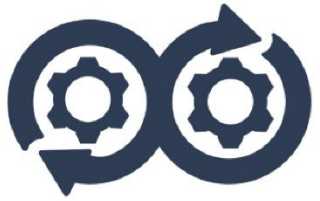
A. On a new virtual machine each time
B. On the same virtual machine reserved for that workflow
C. On a shared server that keeps files between runs
E. On a long-lived VM created by the first run

POP QUIZ:

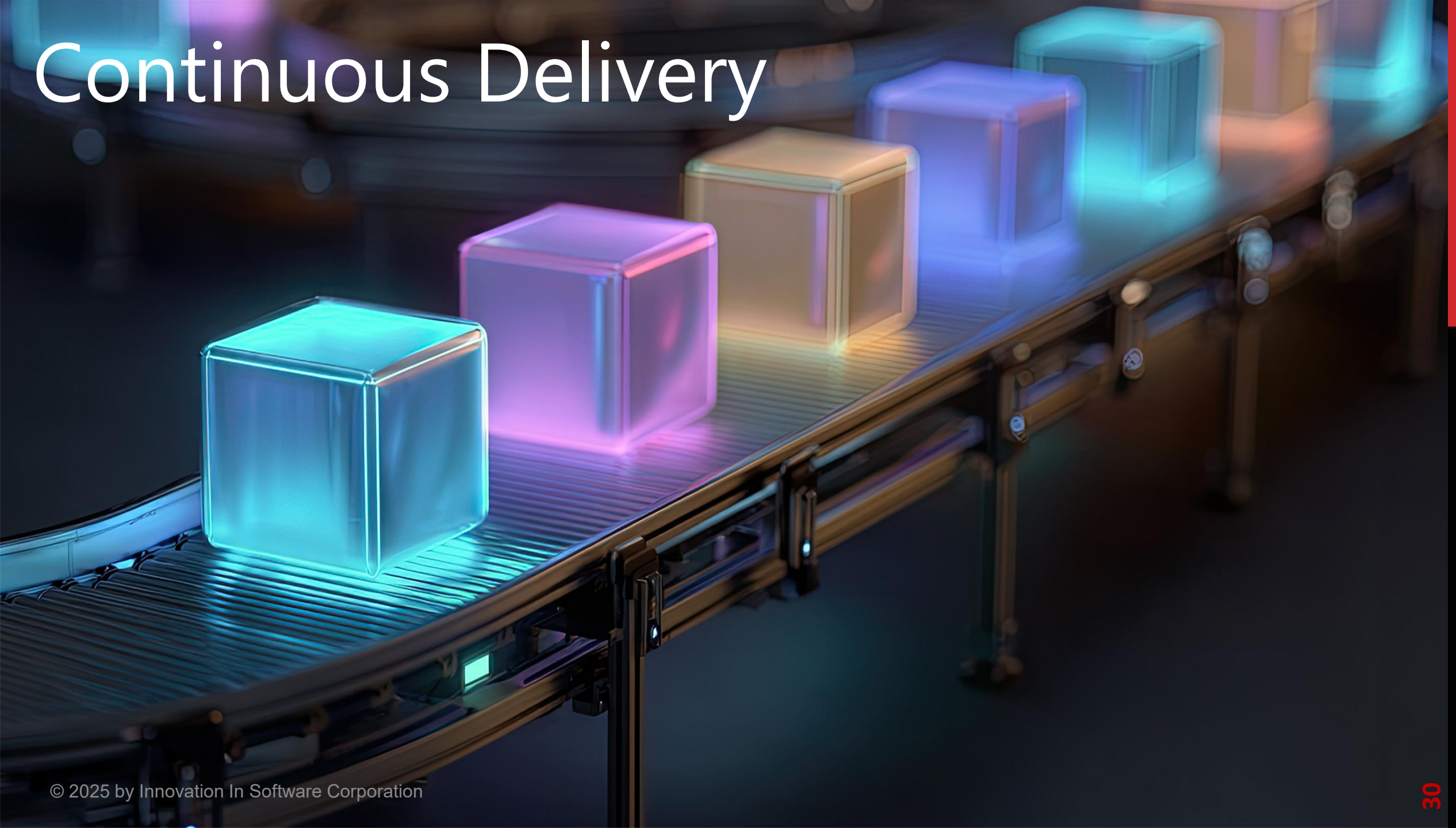
When you run the same GitHub Actions workflow multiple times, where does each run execute?

A. **On a new virtual machine each time**
B. On the same virtual machine reserved for that workflow
C. On a shared server that keeps files between runs
E. On a long-lived VM created by the first run

Lab: GitHub Workflow



Continuous Delivery



Continuous Delivery

Continuous Delivery ensures that every change is automatically built, tested, and packaged so the application is always ready for release. The goal is to remove manual steps from the path between code and a deployable artifact, increasing confidence and consistency.

With Continuous Delivery, teams can release at any time because the system maintains a deploy-ready state. Deployments are still controlled by humans or approval processes, but the pipeline handles all preparation so releases become routine and low risk.

Packaging Software

Software is often bundled into a single artifact so it can be deployed consistently across environments. Packaging separates the application from the environment, making deployment predictable and repeatable.

- Creates one build that works in dev, test, and production
- Allows configuration to be injected at runtime
- Reduces drift between environments
- Simplifies rollbacks and version management
- Forms the foundation for container based deployments and AMI based images

Delivering Packages

Continuous Delivery automates the creation of deployable artifacts by packaging software into consistent, environment agnostic builds. These packages can be containers, AMIs, or other bundled formats that behave the same wherever they run.

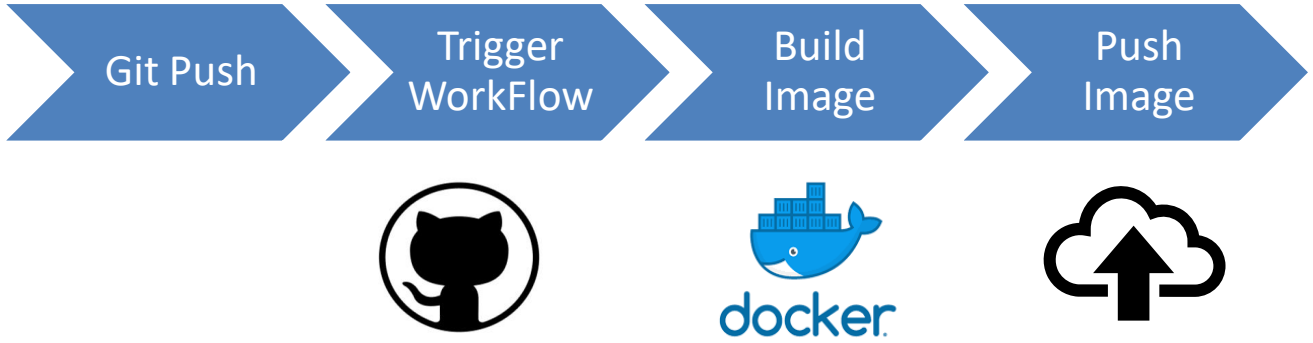
Once built, the pipeline delivers these artifacts to a repository where they are versioned and ready for release. This automated path from code to a stored, deployable package ensures every change is always prepared for deployment with minimal manual effort.

Basics of Images and Containers

Images package an application and everything it needs to run into one self-contained build. Containers then run that image the same way in any environment, which makes them ideal for automation pipelines.

- Guarantees consistent behavior across dev, test, and production
- Makes packaging and delivery predictable and repeatable
- Allows pipelines to build an image once and deploy it anywhere
- Simplifies rollbacks by keeping previous image versions
- Fits naturally into repositories like Docker Hub or ECR

Packaging Software



Continuous Integration

- Push code to the repository
- Workflow starts automatically
- Build the application
- Run tests, linting, and security scans
- Verify the code is stable and ready to package

Continuous Delivery

- Package the application into an image
- Push the image to a repository
- Produce a versioned artifact ready for deployment

Git Actions

A GitHub Action is a reusable automation component that you can plug into a workflow. Instead of writing shell commands yourself, an action performs that work for you.

Key points

- An action is an abstraction that hides low level commands
- Most actions run shell scripts or other actions behind the scenes
- Workflows use actions to perform tasks without writing OS level commands
- Encourages reuse and consistency across pipelines
- Lets teams build complex automation using simple, readable steps

Actions vs Run

No Action

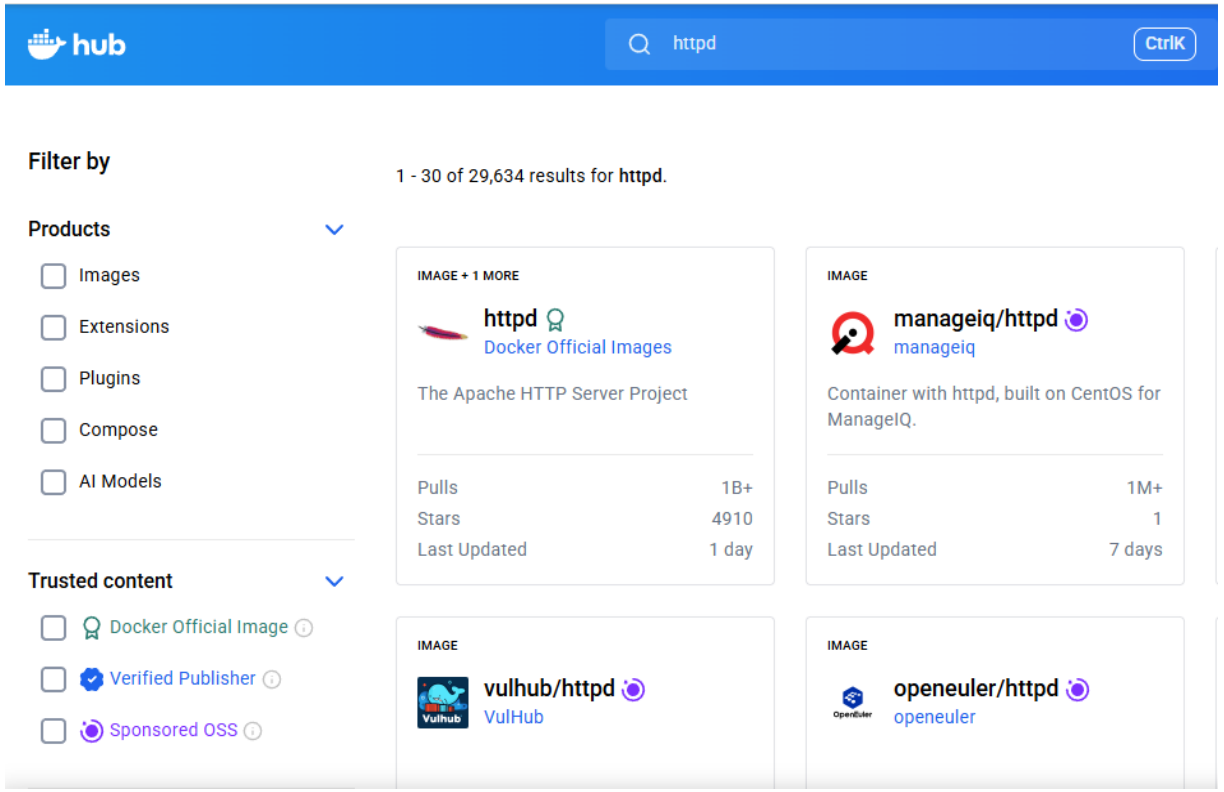
```yaml
steps:
- uses: actions/checkout@v4
- name: Build the Docker image
  run: docker build . --file Dockerfile
```

Action

```yaml
    name: Build and push
    uses: docker/build-push-action@v6
    with:
      push: true
      tags: user/app:latest
```

Many actions from the GitHub Marketplace are maintained by trusted companies like Docker, providing a clean and reliable way to perform common tasks. You can also choose to run the equivalent commands directly on the underlying OS if you want full control.

Docker Hub



Docker Hub is a public registry where companies and individuals store and share container images. It acts as a central location for delivering packaged applications.

- Contains thousands of image repositories from many organizations
- Allows teams to push versioned images for others to pull and use
- Supports public and private repositories
- Serves as a delivery point for CI/CD pipelines that build and publish images

Docker Hub Registries

Registries maintain **tags** that act as versions of an image, allowing you to manage and reference specific builds. You can also upload documentation for each repository and control access by making images public or keeping them private for your organization.

Dockerfile

```
# Use official Python runtime as base image
FROM python:3.11-slim

# Set working directory
WORKDIR /app

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY app/ ./app/

# Expose port
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
    CMD python -c "import requests; requests.get('http://localhost:8000/health')" || exit 1

# Run the application
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

A **Dockerfile** defines how to build your source code into an image artifact. It describes the environment, dependencies, and steps needed so the application can run consistently inside a container.

Dockerfile

```
# Use official Python runtime as base image
FROM python:3.11-slim

# Set working directory
WORKDIR /app

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY app/ ./app/

# Expose port
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
    CMD python -c "import requests; requests.get('http://localhost:8000/health')" || exit 1

# Run the application
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

FROM
Defines the base image your application will build on.

COPY
Adds your source code or files into the image.

RUN
Executes commands during the image build, such as installing dependencies.

CMD
Defines the default command that runs when the container starts.

Containers

- A container stops when its **main process** exits. Docker does not keep containers alive on its own.

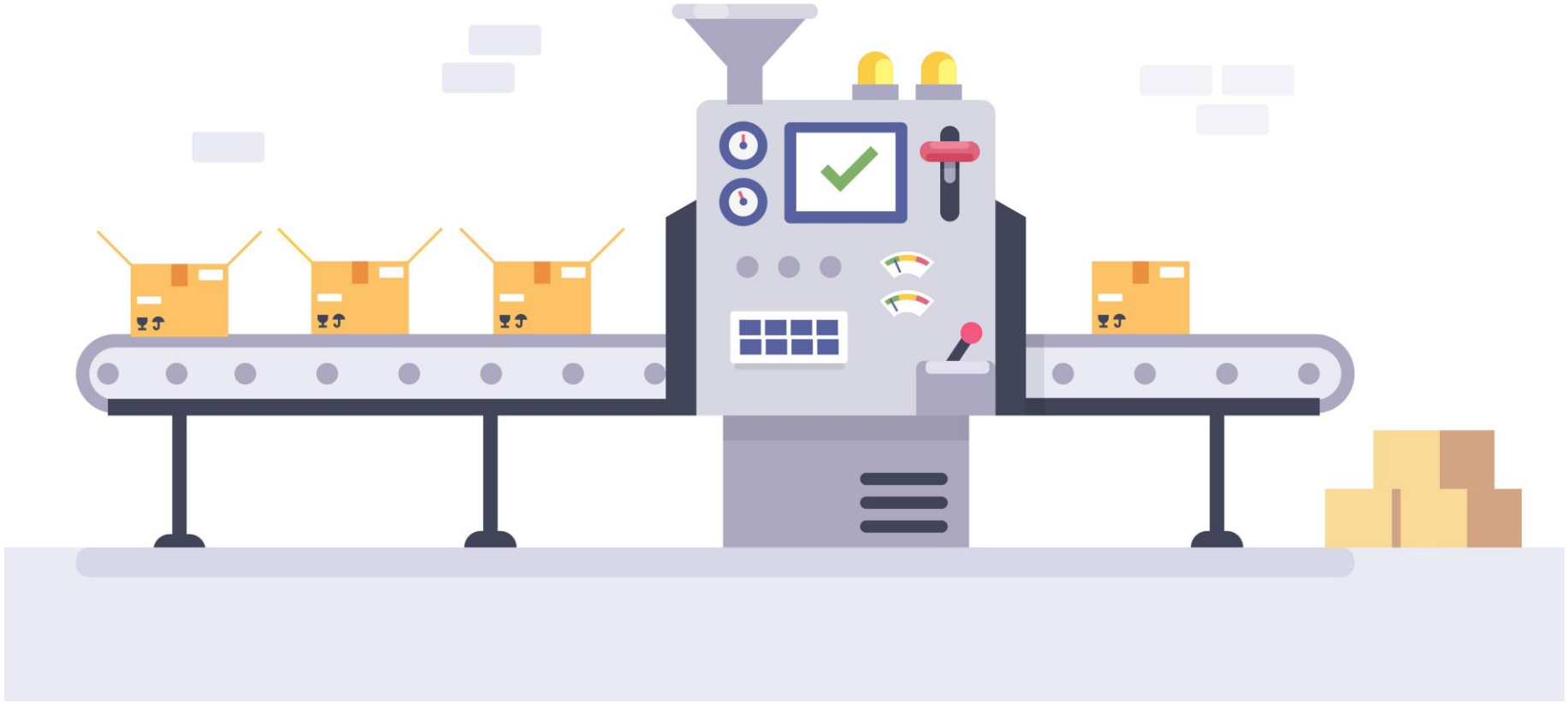
- Containers read **environment variables** at runtime, allowing the same image to run with different configurations in dev, test, or production.

- Images are **immutable**, but containers are not. You configure them through runtime settings, not by changing the image.

- Logs and output come directly from the main process, which makes them easy to capture in a CI/CD pipeline.

- Since images package dependencies, builds are consistent across environments, which is ideal for automated testing and delivery.

CI CD



POP QUIZ:

What does Continuous Delivery focus on in a CI/CD pipeline?

A. Automatically deploying to infrastructure
B. Building, packaging, and storing release-ready artifacts
C. Running pipelines when code is delivered to a repository
D. Continuously delivering tests

POP QUIZ:

What does Continuous Delivery focus on in a CI/CD pipeline?

A. Automatically deploying to infrastructure
B. **Building, packaging, and storing release-ready artifacts**
C. Running pipelines when code is delivered to a repository
D. Continuously delivering tests

POP QUIZ:

Why are Docker images useful for CI/CD pipelines?

A. They run faster than virtual machines
B. They allow the same build to run in any environment
C. They eliminate the need for testing
D. They automatically deploy to docker hub

POP QUIZ:

Why are Docker images useful for CI/CD pipelines?

A. They run faster than virtual machines
B. **They allow the same build to run in any environment**
C. They eliminate the need for testing
D. They automatically deploy to docker hub

POP QUIZ:

What is Docker Hub considered?

A. A cloud compute engine
B. A container runtime
C. A container image registry
D. A virtual machine marketplace

POP QUIZ:

What is Docker Hub considered?

A. A cloud compute engine
B. A container runtime
C. **A container image registry**
D. A virtual machine marketplace

POP QUIZ:

When pushing an image, what must the workflow do first?

A. Run Terraform apply
B. Pull the image from Docker Hub
C. Build the image from the Dockerfile
D. Create a Kubernetes deployment

POP QUIZ:

When pushing an image, what must the workflow do first?

A. Run Terraform apply
B. Pull the image from Docker Hub
C. **Build the image from the Dockerfile**
D. Create a Kubernetes deployment

POP QUIZ:

In a GitHub Action, what does the `uses:` keyword typically indicate?

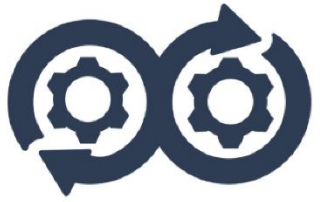
A. Running a shell command
B. Importing a reusable action
C. Starting a new virtual machine
D. Pushing secrets to Docker Hub

POP QUIZ:

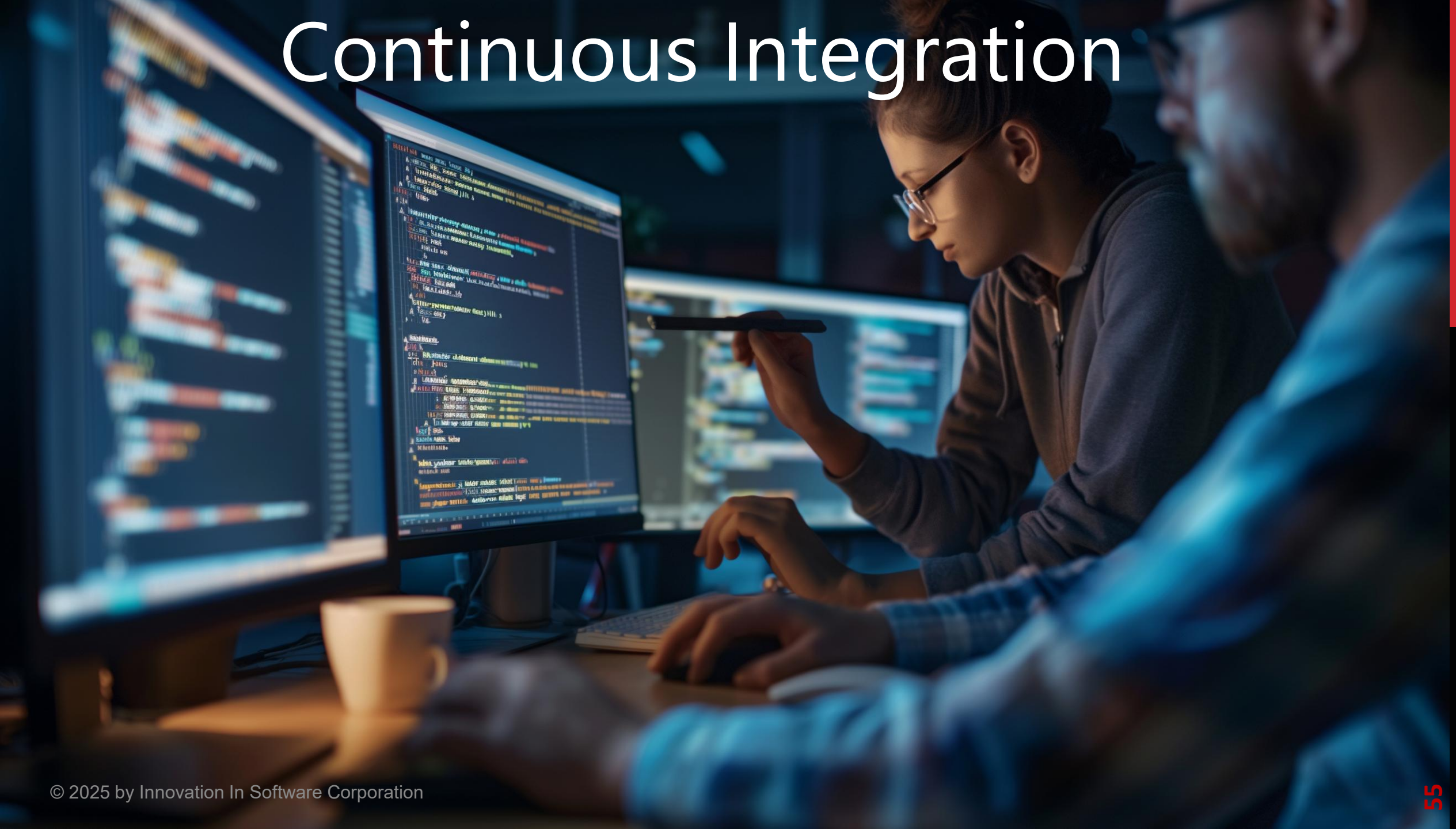
In a GitHub Action, what does the `uses:` keyword typically indicate?

A. Running a shell command
B. **Importing a reusable action**
C. Starting a new virtual machine
D. Pushing secrets to Docker Hub

Lab: GitHub Workflow - Docker



Continuous Integration



Continuous Integration

Continuous Integration is more than just running a workflow when code is pushed. It is the entire automated phase that ensures every change is valid, stable, and safe before it moves toward delivery.

Key parts of CI

- Automatically building the application
- Running tests to verify functionality
- Performing linting for code quality
- Running security scans and dependency checks
- Ensuring the codebase stays in a deployable state

Continuous Integration

Unit Testing in CI

Unit tests validate small, isolated parts of the application to ensure they behave correctly. In a CI pipeline, these tests run automatically to catch issues early.

Key points

- Writing and maintaining unit tests is the responsibility of developers
- Test Driven Development (TDD) creates tests **before** writing the implementation
- Senior engineers often define the tests, and developers write code that satisfies them
- Strong unit testing improves stability and speeds up integration

Security in CI

Security scanning checks your code and dependencies for risks before anything is delivered. It helps catch issues early in the pipeline instead of in production.

What scanners look for

- Exposed secrets such as API keys or credentials
- Vulnerabilities in dependencies and packages (CVEs)
- Misconfigurations in Dockerfiles or infrastructure code
- Outdated or risky library versions

Common tools

- **Trivy** for vulnerabilities, secrets, and config scanning
- **GitHub Advanced Security** with secret scanning and Dependabot alerts
- **Snyk** for dependency and container scanning
- **Aqua / Prisma / Anchore** for enterprise policies

Linting in CI

Linting and formatting help maintain clean, readable code across a team. While less critical than testing or security, they play a major role in long term maintainability.

Why it matters

- Encourages consistent coding style across the whole project
- Makes code easier to read, review, and debug
- Prevents overly clever or compressed code that others cannot maintain
- Helps catch simple issues early, like unused variables or syntax errors

POP QUIZ:

What is the primary goal of Continuous Integration?

A. Automatically deploy to production or staging envs
B. Ensure every code change is built, tested, and validated
C. Replace developers with automation
D. Build Docker images

POP QUIZ:

What is the primary goal of Continuous Integration?

A. Automatically deploy to production or staging envs
B. **Ensure every code change is built, tested, and validated**
C. Replace developers with automation
D. Build Docker images

POP QUIZ:

What does a unit test check?

A. The entire application end to end
B. A small, isolated piece of code
C. The health of a running application
D. Cloud infrastructure configuration

POP QUIZ:

What does a unit test check?

A. The entire application end to end
B. **A small, isolated piece of code**
C. The health of a running application
D. Cloud infrastructure configuration

POP QUIZ:

Security scanners commonly look for which of the following?

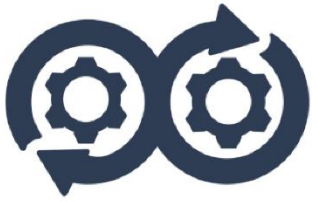
A. Incorrect indentation
B. Exposed secrets or known vulnerabilities
C. Outdated YAML syntax
D. Healthy CPU utilization

POP QUIZ:

Security scanners commonly look for which of the following?

A. Incorrect indentation
B. **Exposed secrets or known vulnerabilities**
C. Outdated YAML syntax
D. Healthy CPU utilization

Lab: GitHub Workflows Enhancements



OIDC Authentication

OIDC

Secrets in Pipelines

Secrets allow pipelines to authenticate into external services such as Docker Hub, AWS, or other APIs. They enable automated workflows to push images, access cloud resources, or fetch protected data.

Secrets in Pipelines

Challenges and risks

- They can be reused in unexpected systems if not scoped properly
- Rotating and maintaining secrets becomes difficult over time
- Storing long-lived credentials increases the attack surface
- A compromised secret grants the same access as a real user

This leads into why OIDC is preferred for secure, keyless authentication.

OIDC

OpenID Connect (OIDC) is an identity layer built on top of OAuth 2.0 that allows secure, short-lived authentication without storing long-term secrets. It lets services verify *who* is requesting access using trusted identity providers like GitHub, AWS, or Google.

In CI/CD, OIDC enables pipelines to authenticate to cloud services without credentials, replacing static keys with secure, temporary tokens issued only when a workflow runs.

OIDC GitHub Access to AWS

As an example, AWS can be configured to trust only a specific GitHub organization or even a single repository when issuing temporary credentials through OIDC. This ensures that only approved workflows can authenticate.

Key benefits

- Only the trusted repository is allowed to request credentials
- No long-lived secrets exist for attackers to steal
- Temporary credentials are issued only during a workflow run
- Access is based on identity verification, not stored keys

This demonstrates how OIDC can provide secure, controlled access in CI/CD pipelines.

OIDC GitHub Access to AWS

AWS manages access through **IAM (Identity and Access Management)**.
A basic approach is to create an IAM user, attach permissions through groups or policies, and generate access keys for that user.

Why this is not secure

- Keys are long lived and can be stolen
- Keys can be reused in any environment, not just CICD
- Hard to rotate and manage safely
- Provides unnecessary attack surface in pipelines

This is why we move away from users and static keys for automation.

OIDC GitHub Access to AWS

A more secure method is to use an **IAM Role** with the minimum permissions needed. A role does not use static keys; instead, it is "assumed" by whoever is trusted to use it.

Key concepts

- A **trust policy** defines *who* is allowed to assume the role
- The role's permissions define *what* it is allowed to do
- We can allow our GitHub OIDC provider to assume the role
- And restrict it to only our organization or specific repository

This creates a tight, secure boundary for CI/CD access to AWS.

GitHub Workflow Artifacts

Pipelines run on ephemeral machines, so files created during a workflow do not persist after the job ends. GitHub provides **artifacts** to store important files from a run so they can be accessed later.

Key points

- Artifacts preserve files even after the runner is destroyed
- They can be downloaded from the workflow run summary
- Useful for logs, test results, build outputs, or packaged files
- Artifacts remain available for a limited retention period

POP QUIZ:

Which of the following is true about OIDC compared to static keys?

A. OIDC tokens never expire
B. OIDC requires storing passwords in GitHub
C. OIDC eliminates long-lived secrets that can be stolen
D. OIDC only works with private repositories

POP QUIZ:

Which of the following is true about OIDC compared to static keys?

A. OIDC tokens never expire
B. OIDC requires storing passwords in GitHub
C. **OIDC eliminates long-lived secrets that can be stolen**
D. OIDC only works with private repositories

POP QUIZ:

What does a trust policy (conceptually) define?

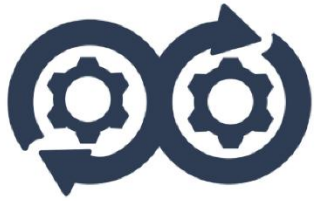
A. If a pipeline is allowed to run on a service
B. Who is allowed to assume a role or identity
C. Who is allowed to connect via OIDC
D. Who is allowed to access an artifact

POP QUIZ:

What does a trust policy (conceptually) define?

A. If a pipeline is allowed to run on a service
B. **Who is allowed to assume a role or identity**
C. Who is allowed to connect via OIDC
D. Who is allowed to access an artifact

Lab: GitHub OIDC



Congratulations

Congratulations on finishing Day 1.
You built the core foundations of modern CI/CD.

Today you learned

- How CI and CD function in a workflow
- How to build and package software into images
- How testing, linting, and security scanning fit into CI
- How registries store versioned artifacts
- Why OIDC provides secure, keyless authentication