# Continuous Integration/Delivery Basics

“The most powerful tool we have as developers is automation.”
— Scott Hanselman

# CI/CD: The what, why, and how

- Prerequisites
- Development Workflow
- CI/CD Explained
- Why CI/CD?
- What makes CI/CD successful
- CI/CD Tools
- GitHub Actions

# Prerequisites

- Understanding of Terminal basics
- Understanding of project build automation tools (Gradle, Maven, npm)
- YAML – is a powerful instrument that becomes a meme in programming world

# Development Workflow

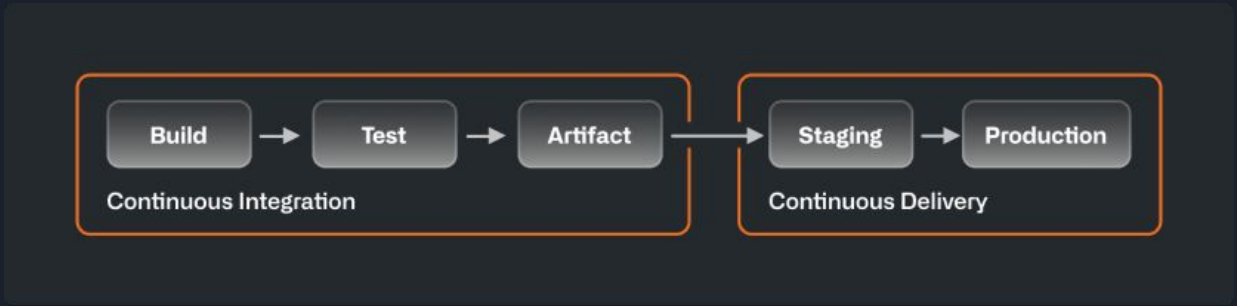
- Develop
- Build
- Test (Run all tests)
- Commit
- Push
- Review
- Merge
- Deploy

# CI/CD Explained

# CI/CD explained

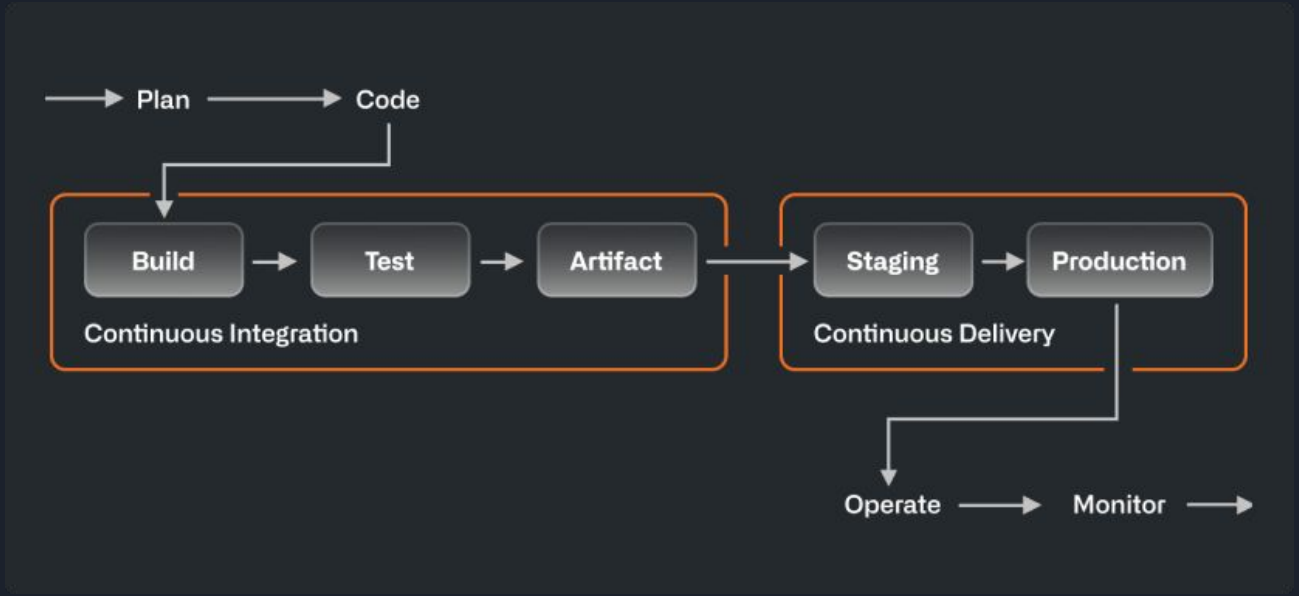
| Continuous integration (CI) | Continuous delivery (CD) | Continuous deployment (CD) |
| --- | --- | --- |
| automatically builds, tests, and integrates code changes within a shared repository; then | automatically delivers code changes to production-ready environments for approval; or | automatically deploys code changes to customers directly. |

# A CI/CD pipeline




https://resources.github.com/ci-cd/

# Continuous delivery vs. continuous deployment




https://resources.github.com/ci-cd/

# Why CI/CD?

### Development velocity

Ongoing feedback allows developers to commit smaller changes more often, versus waiting for one release.

### Stability and reliability

Automated, continuous testing ensures that codebases remain stable and release-ready at any time.

### Business growth

Freed up from manual tasks, organizations can focus resources on innovation, customer satisfaction, and paying down technical debt.

# What makes CI/CD successful

## Automation

CI/CD *can* be done manually—but that's not the goal. A good CI/CD workflow automates builds, testing, and deployment so you have more time for code, not more tasks to do.

## Transparency

If a build fails, developers need to be able to quickly assess what went wrong and why. Logs, visual workflow builders, and deeply integrated tooling make it easier for developers to troubleshoot, understand complex workflows, and share their status with the larger team.

## Speed

CI/CD contributes to your overall DevOps performance, particularly speed. DevOps experts gauge speed using two DORA metrics: Lead time for changes (how quickly commits are made to code in production) and deployment frequency (how often you commit code).

# What makes CI/CD successful

## Resilience

When used with other approaches like test coverage, observability tooling, and feature flags, CI/CD makes software more resistant to errors. DORA measures this stability by tracking mean time to resolution (how quickly incidents are resolved) and change failure rate (the number of software rollbacks).

## Security

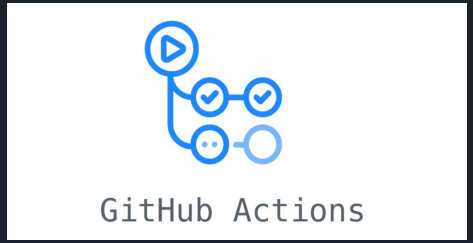
Automation includes security. With DevSecOps gaining traction, a future-proof CI/CD pipeline has checks in place for code and permissions, and provides a virtual paper trail for auditing failures, security breaches, non-compliance events.

## Scalability

CI/CD isn't just about automation; it's also about ensuring scalability. A robust CI/CD setup should effortlessly expand with your growing development team and project complexity. This means it can efficiently handle increased workloads as your software development efforts grow, maintaining productivity and efficiency.

# CI/CD Tools

CI/CD


Jenkins



GitHub Actions

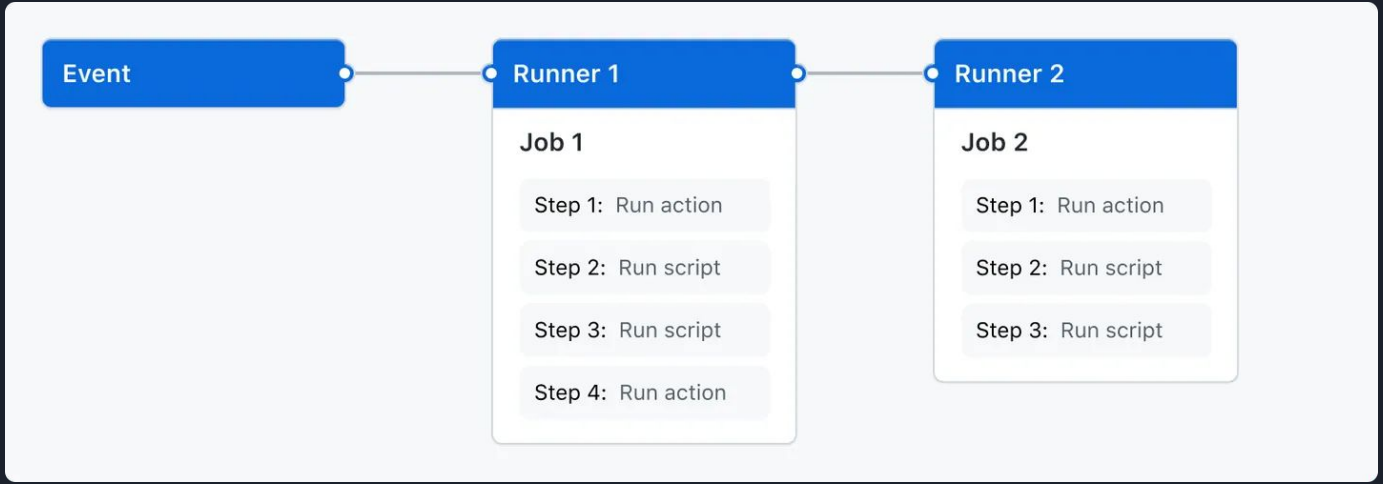
# Overview

- GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.

- GitHub Actions goes beyond just DevOps and lets you run workflows when other events happen in your repository. For example, you can run a workflow to automatically add the appropriate labels whenever someone creates a new issue in your repository.

- GitHub provides Linux, Windows, and macOS virtual machines to run your workflows, or you can host your own self-hosted runners in your own data center or cloud infrastructure.

# The components of GitHub Actions



# Workflows

- A workflow is a configurable automated process that will run one or more jobs.

- Workflows are defined by a YAML file checked in to your repository and will run:
  - when triggered by an event in your repository
  - can be triggered manually
  - at a defined schedule

- Workflows are defined in the `.github/workflows` directory in a repository
- Repository can have multiple workflows (each can perform different set of tasks)

# Events

- An event is a specific activity in a repository that triggers a workflow run

- Examples:
  - someone creates a pull request
  - someone opens an issue
  - someone pushes a commit to a repository

- Also we can trigger run by:
  - A schedule
  - Posting to a REST API call
  - manually

# Jobs

- A job is a set of steps in a workflow that is executed on the same runner
- Each step is either a *shell script* that will be executed, or an *action* that will be run
- Steps are executed in order and are dependent on each other
- Since each step is executed on the same runner, you can share data from one step to another

- Example:

  We have a step that builds our classes followed by a step that tests the classes that were built.

# Actions

- An *action* is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task

- Usage of action can help to reduce the amount of repetitive code that you write in your workflow files

- Examples of actions:
  - pull your git repository from GitHub
  - set up the correct toolchain for your build environment

- We can write our own actions, or we can find actions to use in our workflows in the GitHub Marketplace.

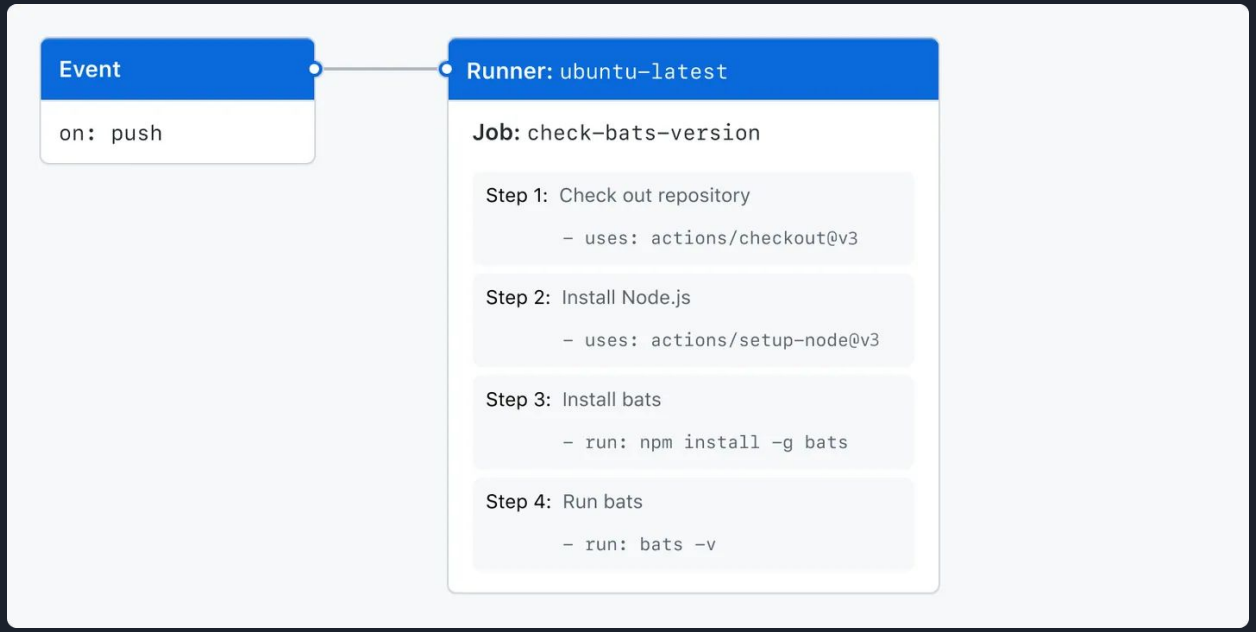
# Runners

- A runner is a server that runs your workflows when they're triggered
- Each runner can run a single job at a time
- GitHub provides Ubuntu Linux, Microsoft Windows, and macOS runners to run your workflows
- Each workflow run executes in a fresh, newly-provisioned virtual machine

# Workflow Example

```yaml
name: learn-github-actions
run-name: ${{ github.actor }} is learning GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bat
      - run: bats -v
```

# Visualizing the workflow file



# Examples

- Any GitHub repo (`lp4k`, `spring-framework-examples`)
- GeekHub Template repository walkthrough

# Environment Variables and Secrets

- Variables provide a way to store and reuse non-sensitive configuration information
- You can store any configuration data such as compiler flags, usernames, or server names as variables
- Variables are interpolated on the runner machine that runs your workflow
- Commands that run in actions or workflow steps can create, read, and modify variables

- By default, variables render unmasked in your build outputs.
- If you need greater security for sensitive information, such as passwords, use **secrets** instead -> Using secrets in GitHub Actions

# Environment Variables Example

```yaml
name: Greeting on variable day

on:
  workflow_dispatch

env:
  DAY_OF_WEEK: Monday

jobs:
  greeting_job:
    runs-on: ubuntu-latest
    env:
      Greeting: Hello
    steps:
      - name: "Say Hello Mona it's Monday"
        run: echo "$Greeting $First_Name. Today is $DAY_OF_WEEK!"
        env:
          First_Name: Mona
```

# Useful materials

- Understanding GitHub Actions
- GitHub Actions Tutorial - Basic Concepts and CI/CD Pipeline with Docker
- Variables
- Using secrets in GitHub Actions