CI/CD Fundamentals & GitHub Actions Course

Module 1: CI/CD Foundation

What is CI/CD?

Continuous Integration (CI) is the practice of automatically integrating code changes from multiple developers into a shared repository frequently - typically multiple times per day. Each integration triggers automated builds and tests to detect integration errors quickly. It is like having a quality control checkpoint every time a worker adds a part to the assembly line. Instead of waiting until the end to find problems, you catch them immediately.

Continuous Deployment/Delivery (CD) extends CI by automatically deploying code changes to production (Continuous Deployment) or preparing them for production deployment (Continuous Delivery). It is like having an automated system that moves the finished product directly to the shipping dock once it passes all quality checks.

The Problem CI/CD Solves

Imagine you're working on a team of 10 developers. Without CI/CD:

- Everyone works on their own copy of the code for weeks
- When it's time to combine everyone's work, conflicts arise
- Manual testing takes days
- Deployments happen late at night with fingers crossed
- When something breaks, it's hard to know what caused it.

Why CI/CD Matters

Traditional software development involved long development cycles, manual testing, and risky deployments. CI/CD solves critical problems:

- **Reduced Integration Hell**: Frequent small integrations prevent massive merge conflicts. Code is integrated multiple times daily.
- **Faster Feedback**: Bugs are caught within minutes, not weeks. Tests run automatically on every change and problems are caught within minutes, not weeks.
- **Reliable Releases**: Automated processes reduce human error. Deployments happen safely during business hours.
- **Faster Time-to-Market**: Features reach users quickly and safely.

Core CI/CD Principles

Build Once, Deploy Everywhere: Create artifacts once and promote them through environments without rebuilding. This ensures consistency and reduces variables between environments.

Fail Fast: Detect problems early in the pipeline when they're cheaper to fix. A failed unit test should stop the pipeline before expensive integration tests run.

Everything as Code: Infrastructure, deployment scripts, and pipeline configurations should be version-controlled and reviewable.

The CI/CD Mindset Shift

The biggest mindset shift is moving from "big bang" releases to "small, frequent changes." Instead of working in isolation for weeks, you integrate your work daily. This feels scary at first but becomes liberating once you experience it.

Three Core Principles to Remember:

1. **Integrate Early, Integrate Often** - Merge your changes to the main branch at least daily
2. **Automate Everything** - If a human has to remember to do it, it will eventually be forgotten
3. **Fail Fast** - Find problems quickly when they're cheap to fix

Your First Mental Model

Think of CI/CD as a pipeline with gates:

- **Gate 1**: Code Quality (Does it meet our standards?)
- **Gate 2**: Functionality (Does it work?)
- **Gate 3**: Integration (Does it work with everything else?)
- **Gate 4**: Deployment (Can users access it safely?)

Each gate must pass before moving to the next. If any gate fails, the process stops and the developer gets immediate feedback.

Module 2: CI/CD Pipeline Architecture

Key pipeline stages of a Continuous Integration (CI) pipeline architecture.

The exact implementation can vary based on the project, technology stack, and organizational needs, a robust CI pipeline generally follows these core stages:

Continuous Integration Pipeline Stages

Here's a breakdown of the typical stages, focusing on best practices for speed, reliability, and security:

1. **Source Code Management (SCM) Integration / Trigger:**
    - **Purpose:** This is the starting point of the pipeline. It monitors the version control system (e.g., Git, Mercurial) for changes.
    - **Details:**
        - **Webhooks:** The most common method. A webhook is configured in the SCM to trigger the CI pipeline whenever code is pushed to a designated branch (e.g., `main`, `develop`, feature branches).
        - **Polling:** Less efficient, but sometimes used for legacy systems. The CI server periodically checks the SCM for new commits.
        - **Branch Filtering:** Pipelines are often configured to run only for specific branches or to have different behaviors based on the branch.
2. **Build Stage:**
    - **Purpose:** Compiles the source code into executable artifacts.
    - **Details:**
        - **Dependency Resolution:** Downloads and caches external libraries and dependencies (e.g., Maven, npm, pip, Go modules).
        - **Compilation:** Compiles the code using the appropriate compiler for the language (e.g., `javac`, `gcc`, `go build`).
        - **Artifact Generation:** Creates the deployable artifacts (e.g., JARs, WARs, Docker images, executables, binaries). These artifacts should be immutable and ideally tagged with the commit hash or a unique build number.
        - **Static Code Analysis (Pre-build/Early Scan):** Often integrated here to catch basic syntax errors, formatting issues, and some common vulnerabilities *before* full compilation (e.g., Linters, Prettier, SonarQube).
3. **Unit Testing Stage:**
    - **Purpose:** Runs automated unit tests to verify the smallest testable parts of an application.
    - **Details:**
        - **Isolation:** Unit tests should run quickly and in isolation, without external dependencies (databases, network calls).
        - **Code Coverage:** Tools are often integrated to measure code coverage, ensuring a high percentage of the codebase is covered by tests.

        - **Fast Feedback:** This stage is critical for providing immediate feedback to developers on the correctness of their changes. Failure here should stop the pipeline immediately.
4. **Static Application Security Testing (SAST) / Code Quality Analysis:**
    - **Purpose:** Analyzes the source code for security vulnerabilities, coding standard violations, and architectural flaws without executing the code.
    - **Details:**
        - **Security Scans:** Identifies common vulnerabilities like SQL injection, cross-site scripting (XSS), insecure direct object references (IDOR), etc. (e.g., SonarQube, Checkmarx, Fortify).
        - **Code Quality Metrics:** Measures complexity, duplications, and adherence to coding standards.
        - **Early Detection:** Catching these issues early in the pipeline is significantly cheaper and easier to fix than later in the development cycle.
5. **Artifact Storage / Repository:**
    - **Purpose:** Stores the successfully built and tested artifacts in a secure and versioned repository.
    - **Details:**
        - **Immutable Artifacts:** Once an artifact is built and stored, it should not be modified.
        - **Version Control for Artifacts:** Artifacts are typically tagged with the build number, commit ID, or semantic versioning.
        - **Examples:** Artifactory, Nexus, Google Container Registry (GCR), Google Artifact Registry. This ensures that the exact same artifact that passed CI stages can be promoted to higher environments (e.g., QA, Staging, Production).

Optional but Recommended CI Stages:

- **Integration Testing (often part of CD):** While sometimes considered part of CD, lightweight integration tests can be run in CI to verify interactions between different modules or services in a controlled environment.
- **Component Testing:** Focuses on testing individual components in isolation, but with their external dependencies mocked or stubbed.
- **Dependency Vulnerability Scanning (Software Composition Analysis - SCA):** Scans for known vulnerabilities in third-party libraries and open-source components used by the application (e.g., Snyk, OWASP Dependency-Check). This is crucial for supply chain security.

Key Principles of a CI Pipeline:

- **Fast Feedback Loops:** Each stage should be optimized for speed to provide developers with quick feedback on their changes.

- **Automated Everything:** Manual steps are minimized or eliminated entirely.
- **Idempotency:** Running the pipeline multiple times with the same input should produce the same output.
- **Traceability:** Every artifact, test result, and deployment should be traceable back to the source code commit.
- **Single Source of Truth:** The version control system is the single source of truth for all code and configurations.
- **Shifting Left:** Security and quality checks are moved as early as possible in the development lifecycle.
- **Visibility:** Pipeline status, test results, and build logs are easily accessible to the team.

By implementing these stages, organizations can achieve a robust CI pipeline that ensures code quality, reduces integration issues, and accelerates the delivery of reliable software.

Module 3: GitHub Actions Fundamentals

Automating Your Development Workflow

GitHub Actions is a powerful, flexible, and fully integrated CI/CD (Continuous Integration/ Continuous Delivery) platform that lives right within your GitHub repositories. It allows you to automate virtually any task in your software development lifecycle, from building and testing your code to deploying applications and managing issues.

At its core, GitHub Actions operates on a simple, event-driven model: **when an event happens in your repository, a workflow is triggered to perform a series of automated tasks.**

Key Components of GitHub Actions:

1. **Workflows:**
    - A workflow is an automated process that you define in your repository.
    - They are configured using YAML files (e.g., `my-ci-workflow.yml`) and stored in the `.github/workflows/` directory of your repository.
    - A single repository can have multiple workflows, each tailored for different automation needs (e.g., one for CI, another for deployment, one for issue management).
    - Workflows specify the sequence of jobs to be executed and the events that trigger them.
2. **Events:**
    - An event is a specific activity in your repository that initiates a workflow run.
    - Common events include:
        - `push`: When code is pushed to a branch.
        - `pull_request`: When a pull request is opened, synchronized, or closed.
        - `schedule`: To run workflows at a predefined cron schedule.
        - `workflow_dispatch`: To trigger a workflow manually from the GitHub UI or via API.
        - `issue`: When an issue is opened, labeled, assigned, etc.
    - You define which event(s) trigger your workflow using the `on:` keyword in your YAML file.
3. **Jobs:**
    - A job is a set of steps that execute on the same **runner** (a virtual machine or container).
    - Workflows can contain one or more jobs. By default, jobs run in parallel, but you can define dependencies between them using the `needs:` keyword to ensure sequential execution.
    - Each job runs in a fresh, isolated environment, ensuring that one job's actions don't interfere with another's.
    - You define a job with a unique identifier and specify the operating system it will run on (e.g., `runs-on: ubuntu-latest`).

4. **Steps:**
    - A step is an individual task within a job. Steps are executed sequentially in the order they are defined.
    - Each step can either:
        - Run a shell command (e.g., `run: npm install`).
        - Use a pre-built **action** from the GitHub Marketplace.
5. **Actions:**
    - Actions are reusable units of code that perform specific, often complex, tasks.
    - They are the fundamental building blocks of steps, allowing you to avoid repetitive scripting.
    - You can use actions created by GitHub, the community (available on the GitHub Marketplace), or even create your own custom actions.
    - Examples include `actions/checkout@v4` (to check out your repository code) or `actions/setup-node@v4` (to configure a Node.js environment).
    - Actions are typically referenced using the `uses:` keyword in a step.
6. **Runners:**
    - A runner is the server that executes your workflow jobs.
    - GitHub provides **GitHub-hosted runners** for various operating systems (Linux, Windows, macOS) that are automatically provisioned and managed.
    - You can also set up **self-hosted runners** on your own infrastructure if you have specific hardware or environment requirements, or if you need to run jobs in a private network.
7. **Secrets:**
    - Secrets are encrypted environment variables that allow you to securely store sensitive information (like API keys, tokens, or passwords) within your GitHub repository or organization settings.
    - They are exposed as environment variables to your workflow jobs at runtime, ensuring sensitive data is not hardcoded in your workflow files.

How it Works (Basic Flow):

1. **Define Workflow:** You create a YAML file in `.github/workflows/` that describes your automation logic, including events, jobs, and steps.
2. **Event Trigger:** A specified event (e.g., a `git push`) occurs in your repository.
3. **Workflow Activation:** GitHub detects the event and activates the corresponding workflow.
4. **Job Execution:** GitHub provisions a runner (e.g., an `ubuntu-latest` VM) and starts executing the jobs defined in your workflow.
5. **Step Execution:** Within each job, steps are run sequentially. These steps can execute shell commands or utilize pre-built actions.
6. **Feedback & Artifacts:** You can monitor the workflow's progress in real-time, view logs, and download any generated artifacts (e.g., build outputs, test reports).