Introduction to Continuous Integration (CI)

Continuous Integration (CI) is the practice of **automatically building and testing code** whenever developers push changes to version control. The primary goal is to detect issues early and maintain a stable, reliable codebase throughout the development lifecycle.

01

Developer Commits Code

Changes are pushed to the version control system

02

Jenkins Detects Push

The CI server automatically identifies new commits

03

Builds and Tests Run

Code is compiled and automated tests execute

04

Alerts on Outcome

Team receives immediate feedback on success or failure

Why CI Matters for Developers

Before CI

Manual Builds

Time-consuming compilation processes prone to human error

Late Bug Discovery

Issues found weeks after code was written, making fixes expensive

"Works on My Machine"

Environment inconsistencies causing deployment failures

With CI

Early Failure Detection

Problems identified within minutes of commit

Faster Feedback Loops

Developers receive immediate validation of their work

Consistent Builds

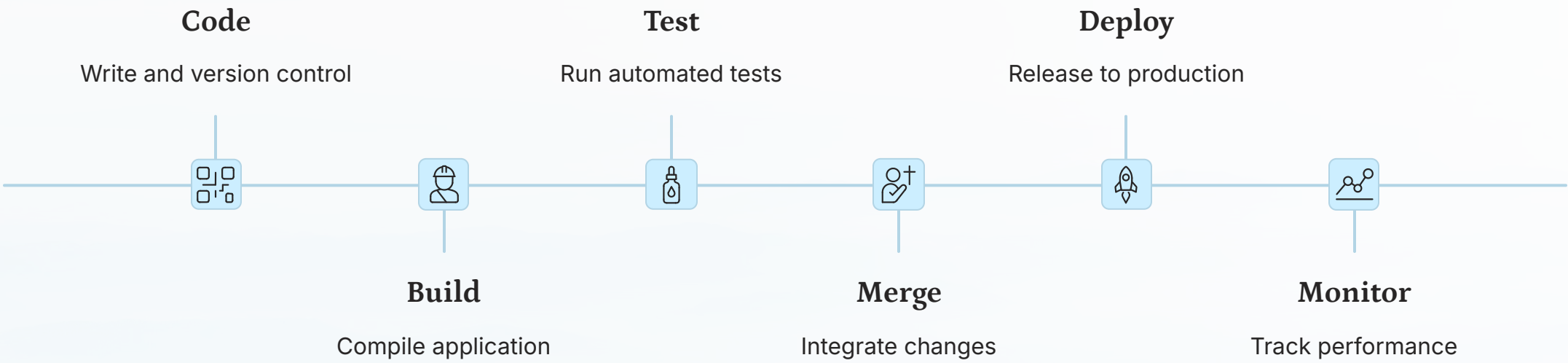
Standardised environments eliminate configuration drift

Merge Confidence

Teams can integrate changes rapidly without fear

CI/CD in the DevOps Lifecycle

Continuous Integration forms the **first half of the CI/CD pipeline**, establishing the foundation for automated software delivery. Understanding where CI fits within the broader DevOps lifecycle is essential for implementing effective automation strategies.



Continuous Integration (CI)

Encompasses **Build + Test + Merge** phases, ensuring code quality before integration

Continuous Delivery/Deployment (CD)

Covers **Deploy + Deliver** stages, automating release to production environments

Jenkins automates the **CI portion** of this pipeline, integrating seamlessly with Git, testing frameworks, and other essential DevOps tools.

Jenkins Overview

Jenkins is an open-source automation server that has become the industry standard for implementing Continuous Integration and Continuous Delivery workflows. Its flexibility and extensive ecosystem make it the tool of choice for development teams worldwide.

Primary Uses

Continuous Integration

Automatically build and validate code changes as they're committed to version control

Build Automation

Compile source code, manage dependencies, and create deployable artefacts

Test Execution

Run unit tests, integration tests, and quality checks automatically

Deployment

Orchestrate releases to staging and production environments

Key Features

- **Plugin ecosystem:** Over 1,800 plugins extending functionality for virtually any tool or platform
- **Pipeline-as-code:** Define builds using Jenkinsfile stored in version control
- **Integration capabilities:** Native support for Git, Docker, AWS, Kubernetes, and more

Jenkins Architecture

Understanding Jenkins' architecture is crucial for designing scalable and efficient CI/CD systems. The master-agent model enables distributed builds and workload management across multiple machines.

Master Node

Orchestrates pipelines, manages scheduling, serves the web interface, and coordinates all build activities. The master delegates actual build execution to agents.

Agent Nodes

Execute build jobs as directed by the master. Agents can run on different operating systems and contain specific tools or environments required for builds.

Executor

A thread on an agent that runs individual builds. Each agent can have multiple executors, enabling parallel build execution.

Visual Flow

1

Git Commit

Developer pushes code

2

Jenkins Master

Detects change and schedules build

3

Build Agent

Executes pipeline steps

4

Reports Back

Results sent to master and team

Installing Jenkins on Ubuntu

Getting Jenkins up and running on Ubuntu is straightforward with a few terminal commands. This installation method uses the official Jenkins repository to ensure you receive regular updates and security patches.

Installation Steps

```
sudo apt update
sudo apt install openjdk-17-jdk -y
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > \\
  /etc/apt/sources.list.d/jenkins.list'
sudo apt update
sudo apt install jenkins -y
sudo systemctl start jenkins
sudo systemctl enable jenkins
```

Important Notes

- Jenkins requires Java 11 or 17 to run properly
- The installation automatically creates a 'jenkins' system user
- Default port is 8080 – ensure this port is open in your firewall
- Enable Jenkins service to start automatically on system boot

Access Jenkins

Once installation completes, access Jenkins in your browser at:

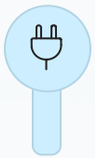
http://<server-ip>:8080

Initial Jenkins Setup

After installation, Jenkins requires a one-time configuration process to secure the instance and prepare it for use. This initial setup wizard guides you through essential security and plugin configuration steps.

Unlock Jenkins

Open Jenkins in your browser and enter the admin password found at `/var/lib/jenkins/secrets/initialAdminPassword`



Install Plugins

Choose "Install suggested plugins" to get essential functionality including Git integration, build tools, and notification systems

Create Admin User

Set up your primary administrator account with a secure username and password

Configure Instance

Set up Jenkins home directory and base URL for your installation

Start Creating

Jenkins is now ready to create your first pipeline

Security Tip: Store the initial admin password securely and change it after setup completes. Consider implementing additional security measures such as matrix-based security or integration with LDAP.

Jenkins Dashboard Walkthrough

The Jenkins dashboard serves as your command centre for managing builds, pipelines, and system configuration. Familiarising yourself with the key sections will help you navigate Jenkins efficiently.

New Item

Create jobs, pipelines, folders, or multi-branch projects. This is your starting point for setting up automation workflows.

Build Queue

View currently running builds and those waiting for available executors. Monitor build progress and estimated completion times.

Manage Jenkins

Access global configuration settings including system configuration, plugin management, security settings, and node management.

Credentials

Securely store GitHub tokens, AWS access keys, SSH keys, and other sensitive authentication information used in your pipelines.

Security Best Practice

Avoid using personal tokens directly in scripts. Always use Jenkins credentials manager to store sensitive information securely. This prevents accidental exposure in logs and ensures credentials can be rotated without modifying pipeline code.

Connecting Jenkins with GitHub

Integrating Jenkins with GitHub enables automatic builds triggered by code commits. This connection forms the backbone of your Continuous Integration workflow, ensuring every change is validated immediately.

01

Install Git Plugin

Navigate to Manage Jenkins → Manage Plugins → Available, search for "Git Plugin" and install it. This provides Jenkins with Git functionality.

02

Add GitHub Credentials

Go to Manage Jenkins → Manage Credentials. Add your GitHub username and personal access token (with repo scope enabled).

03

Configure Repository

In your job configuration, set the repository URL:

```
https://github.com/org/sample-app.git
```

Select the credentials you added in the previous step.

04

Choose Build Triggers

Select either "Poll SCM" for periodic checks or configure a GitHub webhook for instant notifications when code is pushed.

Webhook Recommendation

Webhooks provide the fastest feedback loop by notifying Jenkins immediately when commits are pushed. Configure webhooks in your GitHub repository settings pointing to http://<jenkins-url>/github-webhook/

Understanding Build Triggers

Build triggers determine when Jenkins executes your pipelines. Choosing the right trigger strategy impacts feedback speed, resource utilisation, and team productivity. Different projects may benefit from different trigger approaches.

Manual Trigger

Developer clicks "Build Now" in Jenkins. Useful for on-demand builds or testing pipeline changes before automation.

SCM Polling

Jenkins checks the Git repository periodically for changes. Simple to configure but creates unnecessary load with frequent polling.

Webhooks

GitHub notifies Jenkins immediately when commits are pushed. Most efficient method providing instant feedback to developers.

Scheduled Builds

Cron-style timing for nightly builds, reports, or maintenance tasks. Useful for resource-intensive operations outside working hours.

Example Cron Syntax

```
H/15 * * * *  # Every 15 minutes
H 2 * * *     # Once daily at 2 AM (offset by hash)
H H * * 0     # Once weekly on Sunday
```

Pro Tip: Use the `H` symbol instead of specific numbers in cron expressions. Jenkins distributes the load across the time period, preventing all jobs from starting simultaneously and overwhelming the system.

Creating a Freestyle Job

1. Click “New Item” → Freestyle project
2. Link Git repository
3. Add build steps:
    - Execute shell
    - Run tests
    - Archive artifacts
4. Post-build actions: email or deploy

Example build step:

```
mvn clean package
```

Pipeline-as-Code (Jenkinsfile)

Instead of GUI jobs, define pipelines using code.

Example:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'mvn clean package'
      }
    }
    stage('Test') {
      steps {
        sh 'mvn test'
      }
    }
  }
}
```

Commit `Jenkinsfile` to repo — Jenkins detects it automatically.

Build and Test Stages Explained

The core of any Continuous Integration pipeline revolves around two critical phases: building the application and rigorously testing it. These stages work in tandem to validate every code change.

Build Stage

- Compiles source code into executable binaries or deployable packages.
- Packages all necessary artefacts, such as JAR files, WAR files, or Docker images.

Test Stage

- Automatically runs unit, integration, and sometimes end-to-end tests.
- Reports a clear pass/fail status, preventing untested code from progressing further.

This automation ensures that only thoroughly validated and tested builds are considered for subsequent deployment steps, fostering a reliable development workflow.

Artifact Management

Artifacts are the **output** of your build process.

Example: `.jar`, `.war`, `.zip`, or `docker image`

Jenkins Artifact Storage Options:

- Local workspace
- AWS S3
- Artifactory/Nexus

Example archive step:

```
archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true
```

Using Jenkins with Docker

You can containerize builds for consistency.

Example pipeline:

```
pipeline {
  agent {
    docker { image 'maven:3.9.3-eclipse-temurin-17' }
  }
  stages {
    stage('Build') { steps { sh 'mvn clean package' } }
  }
}
```

This ensures builds run in the same environment everywhere.

Parallel and Sequential Stages

Efficient pipeline execution often relies on structuring tasks to run either one after another (sequentially) or simultaneously (in parallel). Understanding these patterns is crucial for optimising build times and feedback loops.

Sequential Execution

This is the default behaviour in Jenkins pipelines. Stages execute strictly one after another, ensuring dependencies are met before proceeding to the next step. Simple and predictable, but can be slow if stages are independent.

Parallel Execution

Allows multiple steps or stages to run concurrently. This significantly reduces overall pipeline execution time, especially when tasks like unit tests, linting, and security scans can happen independently.

The following example demonstrates how to define parallel stages within your Jenkinsfile:

```
pipeline {
  agent any
  stages {
    stage('Tests') {
      parallel {
        stage('Unit Tests') { steps { sh 'npm test' } }
        stage('Lint') { steps { sh 'npm run lint' } }
        stage('Security Scan') { steps { sh 'npm audit' } }
      }
    }
  }
}
```

Leveraging parallelism is key in Continuous Integration (CI) for large repositories, ensuring rapid feedback on code changes by running independent checks simultaneously.

Viewing Build Logs and Reports

After a Jenkins build completes, or while it's in progress, you can access detailed logs by clicking on the build number and selecting "Console Output". This provides an exact, real-time stream of all shell commands executed and their output, crucial for debugging.

For a more modern and user-friendly visualisation of your pipeline's status and logs, consider exploring the **"Blue Ocean" plugin**.

Success

Indicates the build completed without errors, and all tests passed.

Failed

Signals critical errors prevented the build from completing successfully.

In Progress

Shows the build is currently running, awaiting completion.

Handling Build Failures

Understanding and efficiently addressing build failures is critical for maintaining a smooth CI pipeline and ensuring rapid feedback for developers.

1

Common Causes

- **Failing Tests:** Code changes introduce bugs that break unit or integration tests.
- **Wrong Paths:** Incorrect file paths or directory structures prevent compilation or resource loading.
- **Missing Dependencies:** Required libraries or packages are not available in the build environment.
- **SCM Authentication:** Issues with credentials or access rights when interacting with the source code management system.

2

Best Practices for Resolution

Before diving into deep debugging, try these steps:

- **Re-run Build:** Always use the "Replay" option in Jenkins. This ensures the failure is consistent and not a transient issue.
- **Enable Notifications:** Configure email or Slack notifications for build failures to alert the team immediately, enabling quicker response times.

Real-world CI Example

Sample CI flow for a Node.js app:

```
pipeline {
  agent any
  stages {
    stage('Checkout') { steps { git 'https://github.com/org/app.git' } }
    stage('Install') { steps { sh 'npm ci' } }
    stage('Lint')    { steps { sh 'npm run lint' } }
    stage('Test')    { steps { sh 'npm test' } }
    stage('Build')   { steps { sh 'npm run build' } }
    stage('Archive') { steps { archiveArtifacts 'dist/**' } }
  }
}
```

Recap & Expected Outcomes

By the end of Day 2, developers will be able to:

- Explain CI principles and workflow
- Install and configure Jenkins
- Connect GitHub to Jenkins for automated builds
- Automate build & test pipelines
- Archive and track build artefacts

Next Step: Move to Continuous Deployment (Day 3) — automate delivery and releases, bridging the gap between development and production.