Contents

[1. Introduction 2](#_Toc197616144)

[2. What is CI? 3](#_Toc197616145)

[CI Real-World Example: 5](#_Toc197616146)

[3. What is CD (Continuous Delivery / Continuous Deployment)? 5](#_Toc197616147)

[4. Why introduced CI/CD pipeline 8](#_Toc197616148)

[5. Common CI/CD Pipeline stages: 9](#_Toc197616149)

[6. Learn Git Basics 10](#_Toc197616150)

[7. CI/CD Tools 10](#_Toc197616151)

[8. Pipeline Fundamentals Understand 11](#_Toc197616152)

[9. YAML 12](#_Toc197616153)

[10. Common Automated Tasks in Pipeline 14](#_Toc197616154)

# 1. Introduction

CI/CD stands for Continuous Integration and Continuous Delivery/Deployment, and it is a set of modern DevOps practices used to automate and streamline the process of software development, testing and deployment.

1.1 What is a CI/CD Pipeline?

A CI/CD pipeline is an automated sequence of steps that allows developers to:

* Integrate code changes frequently (CI)
* Automatically test.
* Delivery or deploy to production environments (CD) ensuring fast and reliable software delivery.

1.2 Key Concepts

Continuous Integration (CI)

* Developers frequency commit code to a shared repository
* Each commit triggers an automated build and unit testing process
* Detect integration issues early.

Continuous Delivery (CD)

* Builds from CI are automatically pushed to a staging or testing environment
* Manual approval may still be required before production
* Ensures software is always ready for release

Continuous Deployment (CD)

* Extends Continuous Delivery
* Code changes are automatically deployed to production after passing all tests.
* Requires a high level of test automation and confidence.

1.3 Typical Stages of CI/CD Pipeline

1. Source Stage – Triggered code commits or pull requests.

2. Build Stage – Complies the application

3. Test Stage – Runs unit, integration and end-to-end tests

4. Deploy Stage – Pushes code to staging or production

5. Monitor Stage – Monitors performance and errors post-deployment

1.4 Popular CI/CD Tools

* CI Tools: Jenkins, GitHub Actions, GitLab CI, Travis CI, CircleCI
* CD Tools: Spinnaker, ArgoCD, Harness, Octopus Deploy
* Containers & Orchestration: Docker, Kubernetes

1.5 Benefits of CI/CD

* Faster release cycles
* Early bug detection
* Improved code quality
* Reduced manual effort
* Consistent and repeatable deployments

Real World Example:

A personal blog build with react frontend, Python Flask Backend, and Mysql database.

CI/CD Setup:

Tools:

* GitHub – for version control
* GitHub Actions – for CI/Cd automation
* Docker – to Containerize app
* Heroku or Render – for deployment

# 2. What is CI?

Continuous Integration (CI) is a software development practice where developers frequently integrate their code into a shared repository, usually multiple times a day. Each integrate is verified by an automated build and automates tests to detect errors early in the development process.

Why use CI?

1. Catch Bugs Early: By integration code frequently, CI ensures that bugs are detected early when they are easier to fix.

2. Improve Collaboration: Developers can work on the same codebase without conflicts, as CI ensures that the code works together after every change.

3. Speed up Development: Automated testing ensures that code doesn’t break functionality.

4. Faster Feedback Loop: Developers get immediate feedback about their code, allowing them to fix issues quickly and avoid spending time on irrelevant tasks.

5. Reduce Integration Problems: Integration often reduces the risk of having large, difficult-to-manage integration issues down the road.

Current Situation of CI

In today's fast-paced development environment, CI is an essential practice, especially in Agile development teams, DevOps, and when dealing with micro services or large-scale applications. Modern tools like **Jenkins**, **Travis CI**, **CircleCI**, **GitLab CI**, and **GitHub Actions** help automate the process, making it easy to run tests and deploy code as soon as it's merged into the main branch.

Example of CI in Real Life

Let’s take a practical example of a **web development project** for an e-commerce site.

Scenario:

1. **Developer 1** writes new code for a product page.
2. **Developer 2** writes new code for the shopping cart.
3. Both developers push their code to a shared Git repository.

With CI set up:

* Every time code is pushed, a tool like **Jenkins** automatically runs a build on the repository.
* **Automated tests** run to check that the new product page works correctly, and that the shopping cart hasn’t broken anything else (like user sign-in or payment functionality).
* If a test fails, an alert is sent to the developers, who can quickly fix the issue.
* Once tests pass, the code is automatically deployed to a staging server for further manual testing or review.
* When all features are ready, the code can be automatically deployed to production with zero downtime.

This process allows for quick iteration and reduces the risk of shipping broken or untested code.

**GitHub + Jenkins + Selenium (for testing a web app)**

Scenario:

A team is building an **e-commerce website**. Developers are working on different features:

* One adds a payment module
* Another changes the product search
* A third improves UI

Without CI:

* Developers push code once a week
* On integration day, features clash (conflicts, bugs)
* Testing takes days
* Delay in release

With CI:

* Developers commit code daily to GitHub
* A CI tool like **Jenkins** pulls new code
* It automatically:
  + Builds the app
  + Runs unit and UI tests using **Selenium**
  + Sends success/failure reports
* Bugs are caught early
* Faster, safer releases

### **CI Real-World Example:**

Imagine you're working on a **team of developers** building a web app using **React and Node.js**.

1. Each developer pushes their code to **GitHub**.
2. GitHub is connected to **GitHub Actions (CI tool)**.
3. Every push triggers:
   * Code compilation.
   * Unit tests.
   * Linting.
   * Build generation.
4. If the build **passes**, the code can be merged into the main branch.
5. If the build **fails**, the team is notified via email or Slack.

This ensures that **only tested code** goes into production.

# 3. What is CD (Continuous Delivery / Continuous Deployment)?

CD stands for either Continuous Delivery or Continuous Deployment, both of which are practices that build upon CI (Continuous Integration) by automating the release process of applications.

2.1 Continuous Delivery (CD) – Real Example Explained

Continuous Delivery is a DevOps practice where your code changes are automatically tested and packaged and are always in a deployable state.

The final deployment to production is done manually when the team decides it’s ready.

Real World Example: E-commerce Web Application

Let’s say your team is building an **e-commerce website** using:

* **Frontend:** React.js
* **Backend:** Node.js + Express
* **Database:** MongoDB
* **CI/CD Tools:** GitHub + Jenkins + Docker + AWS

Workflow using Continuous Delivery:

1. **Developer Pushes Code**

* A developer adds a new feature: “Apply Coupon on Checkout”.
* Code is committed and pushed to the develop branch in GitHub.

2. **CI Pipeline Triggers**

* **Jenkins** detects the code push.
* It runs:
  + Unit tests
  + Code linting
  + Build process
* If successful, the app is **built into a Docker image**.

3. **CD Pipeline Executes**

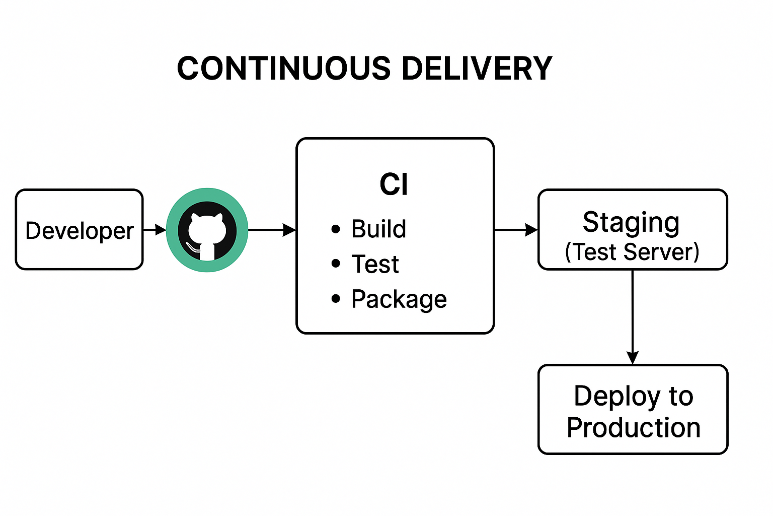
* Jenkins deploys the Docker image to a **staging server**.
* Automated **integration tests** are run on staging.
* QA/testers manually verify the feature in a safe environment.

4. **Ready for Production**

* After validation, the release manager **clicks “Deploy” manually** in Jenkins or AWS CodeDeploy to push the image to production.
* This allows a controlled release, ensuring stability.

Why Continuous Delivery is Useful Here:

| **Benefit** | **Explanation** |
| --- | --- |
| 🚀 Always Ready to Release | Every commit results in a testable version. No more last-minute crunch. |
| 🧪 Safe Testing | Testers can catch issues before they reach users. |
| ⏱️ Manual Control | Production releases happen when you decide, minimizing risk during business hours. |
| 🔁 Quick Fixes | Bugs can be patched and re-tested within minutes. |



2.2 What is Continuous Deployment?

Continuous Deployment (CD) is a software development practice where every change that successfully passes automated tests and integration stages is automatically deployed to the production environment -- without manual intervention.

Key Characteristics:

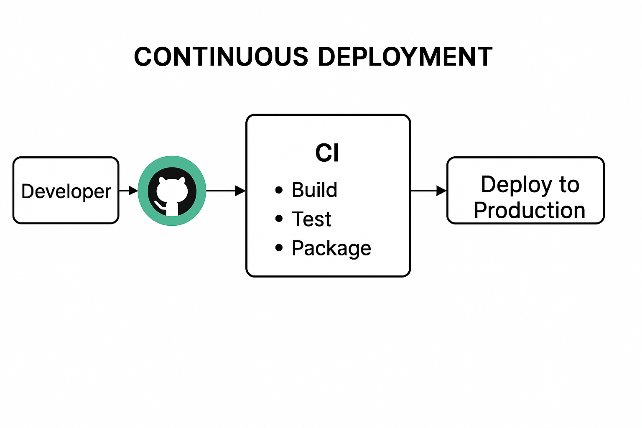
* Automation-first: From code commit to deployment
* Zero-touch delivery: No need for a human to approve or trigger deployment
* Rapid release cycles: Updates, bug fixes, and features go live fast.
* High confidence in tests: Since there’s no manual approval, test coverage must be strong.

Why use Continuous Deployment?

| **Situation** | **Benefit of Continuous Deployment** |
| --- | --- |
| Fast-paced startup | Ship features daily, respond to user feedback fast |
| SaaS applications | Roll out updates/fixes frequently and automatically |
| Global teams with 24/7 releases | Avoid time zone bottlenecks in release cycles |
| DevOps & Agile environments | Enables faster innovation with lower deployment risks |

Real Example: E-commerce Website

1. Developer commits code for a new "Wishlist" feature.
2. Code is pushed to GitHub → triggers CI/CD pipeline.
3. Pipeline performs:
   * **Build**
   * **Unit tests**
   * **Integration tests**
   * **Security scans**
4. If all tests pass:
   * Code is **automatically deployed** to the live website.
   * Users see the new “Wishlist” feature without downtime or manual approval.



# 4. Why introduced CI/CD pipeline

Problem before CI/CD:

1. Manual Builds & Tests – Time-consuming and error prone.

2. Integration Hell – Code from different developers often clashed when merged.

3. Delayed Release – Weeks or months before release.

4. Late detection of bugs (found just before release)

Who introduced the concept?

CI was popularized by Martin Fowler (software engineer & author) and the Extreme Programming (XP) movement in the early 2000s.

Purpose of Introducing CI/CD:

| **Goal** | **Description** |
| --- | --- |

|  |  |
| --- | --- |
| 🚀 Faster Releases | Automate build, test, and deployment to deliver features quickly. |

|  |  |
| --- | --- |
| ✅ Improved Quality | Run automated tests early and often. |

|  |  |
| --- | --- |
| 🤝 Better Collaboration | Integrate code continuously to avoid conflicts and bottlenecks. |

|  |  |
| --- | --- |
| 🔁 Reliable Rollbacks | Revert broken deployments automatically or with minimal effort. |

|  |  |
| --- | --- |
| 📊 Consistent Environments | Use tools like Docker to ensure same behaviour across development to production. |

Real world Example:

 Before CI/CD: Developers at a bank release updates every **6 months**, with lots of bugs and downtime.

 After CI/CD: They deploy updates **daily or weekly**, with better testing and fewer customer issues.

# 5. Common CI/CD Pipeline stages:

A CI/CD pipeline is an automated process that helps developers deliver code changes more frequently and reliable. It usually involves three main stages.

1. Build

* Source code is compiled.
* Dependencies are installed
* Code is pack

Why used?

* Converts code into an executable form.
* Catches compile –time errors early

Real-world Example:

In a React app, npm run build compiles all components and bundles them for deployment.

2. Test

* Automated tests (unit, integration, functional) run.
* Validates the logic and functionality of the application.

Why used?

* Detects bugs before code goes live.
* Increase confidence in releases.

Simple Example

In an **online banking app**, automated tests check that:

* Login works.
* Transfers are processed correctly.
* Account balances update in real-time.

3. Deploy

Code is deployed to staging or production environment

Why used?

 Automates release.

 Ensures consistent environments and fewer manual errors.

Real Example

In a **food delivery app**, new features like “Live Order Tracking” are auto-deployed to production once tests pass.

# 6. Learn Git Basics

Learn Git basics: clone, commit, push, pull, branch, merge

# 7. CI/CD Tools

There are many CI/CD tools available today, each designed to automate building, testing and deploying software efficiently. The choice depends on project scale, team preference, clouyd compatibility, and budget.

Tools

| **Tool** | **Type** | **Highlights** |
| --- | --- | --- |
| **Jenkins** | Open-source | Most flexible; large plugin ecosystem; widely used in enterprise. |
| **GitHub Actions** | Cloud-based | Seamless with GitHub repos; YAML-based; great for open-source projects. |
| **GitLab CI/CD** | Built-in CI/CD | Fully integrated with GitLab; excellent DevOps lifecycle support. |
| **CircleCI** | Cloud & on-prem | Fast performance; great Docker support. |
| **Travis CI** | Cloud-based | Popular with open-source; integrates easily with GitHub. |
| **Azure DevOps** | Cloud/Hybrid | Ideal for .NET/Windows apps; enterprise-grade pipelines. |
| **Bitbucket Pipelines** | Built-in | Simple setup; works well with Atlassian tools (Jira, Bitbucket). |
| **TeamCity** | On-prem/cloud | Developed by JetBrains; rich in features; good for large-scale projects. |
| **AWS CodePipeline** | Cloud-native | Fully integrates with AWS services; good for cloud-native apps. |

# 8. Pipeline Fundamentals Understand

A **pipeline** in CI/CD is like an automated **assembly line for software**. It defines a **sequence of steps** to build, test, and deploy code reliably and consistently.

Basic CI/CD Pipeline Structure

A pipeline is generally structured into **stages**, which are further divided into **jobs** or **steps**.

Pipeline Structure Breakdown:

| **Element** | **Description** |
| --- | --- |
| **Stages** | High-level phases like build, test, deploy. |
| **Jobs** | Group of steps executed together within a stage. May run in parallel. |
| **Steps** | Individual actions (e.g., install packages, run tests). |
| **Triggers** | Events that start the pipeline (e.g., push, pull\_request, schedule). |
| **Runners** | Machines (VMs/containers) that execute your jobs (hosted or self-hosted). |

Common Pipeline Stages

 **Build**

* Compile code, install dependencies.
* Ex: npm install, mvn clean install, docker build.

 **Test**

* Run unit, integration, or security tests.
* Ex: pytest, npm test, jest.

 **Deploy**

* Push artifacts to production or staging.
* Ex: scp, docker push, kubectl apply.

# 9. YAML

In CI/CD (like Github Actions, GitLab CI, Azure Pipelines), YAML files are used to define the workflow – what happens when you push code, like build, test, and deploy steps.

What is YAML?

YAML stands for “YAML Ain’t Markup Language”

It is a human-readable data format, commonly used for configuration files.

Why YAML is used in CI/CD:

| **Benefit** | **Explanation** |
| --- | --- |
| **Readable & Simple** | Easy to understand with indentation and no complex syntax. |
| **Structured Workflow** | Allows defining stages (build, test, deploy) in a clean hierarchy. |
| **Platform Standard** | Most CI/CD tools (GitHub Actions, GitLab, etc.) use YAML to configure flows. |
| **Version Controlled** | Stored in Git (e.g., .github/workflows/ci.yml) with your source code. |
| **Supports Variables** | Allows using environment variables, conditions, matrices, etc. |

Key YAML Concepts

 **Indentation matters** (use 2 spaces).

 **Key-value pairs** (key: value)

 **Lists** are written with -:

steps:

- name: Step 1

run: echo "Hello"

- name: Step 2

run: echo "World"

YAML Syntax Basics

name: My CI Pipeline # Name of the workflow

on: [push] # Trigger: when to run

jobs:

build:

runs-on: ubuntu-latest # Runner type

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.10'

- name: Install dependencies

run: pip install -r requirements.txt

- name: Run tests

run: pytest

# 10. Common Automated Tasks in Pipeline

CI/CD Pipeline Stages Working

1. Build Stage

🔹 What Happens:

* Your source code is compiled or packaged (if needed).
* Dependencies are installed (e.g., pip install -r requirements.txt).
* This turns source code into an **executable format**.

🧠 Why It's Used:

* Ensures that the code **compiles successfully** and all dependencies work.
* Catches early errors (like syntax or missing packages).

2. Test Stage

🔹 What Happens:

* Runs **automated tests**, such as **unit tests**, **integration tests**, etc.
* Verifies code correctness, logic, and functionality.

🧠 Why It's Used:

* Catch bugs **before** they reach users.
* Ensures changes don't break existing code (**regression testing**).

Example

pytest tests/

3. Package Stage

🔹 What Happens:

* Your code is bundled into a **distributable format**, such as:
  + A .whl or .tar.gz file for Python packages
  + A Docker image
  + A .jar file for Java

🧠 Why It's Used:

* Prepares the application for deployment.
* Makes it portable across environments.

Example

python setup.py sdist bdist\_wheel

4. Deploy Stage

🔹 What Happens:

* Pushes your packaged application to **production** or **staging**.
* Often uses tools like scp, kubectl, or docker push.

🧠 Why It's Used:

* Final step that gets your app **live** for users.
* Automates the release, reducing human error and speeding delivery.

Example

scp my\_app.tar.gz user@server:/apps/

10.1 Common Automated Tasks in Pipeline

1. Unit Testing (Why use It)?

Checks if individual parts of the code work as expected.

Automated unit tests help catch issues early and often.

Example

- name: Run Unit Tests

run: pytest

2. Code Linting (Why use it)?

Automatically checks code quality and style.

Prevents inconsistent code, bad patterns or errors.

Example

- name: Lint with Flake8

run: flake8 your\_project/

3. Build & Compile (Why use It)?

Converts source code into a runnable format.

Makes sure all dependencies are installed.

Catches build-time error (especially important for complied languages).

Example

- name: Install & Build

run: |

pip install -r requirements.txt

python setup.py build