Lab # 23

CI/CD: What, Why, and How?
Continuous Integration / Continuous Development

Objective:

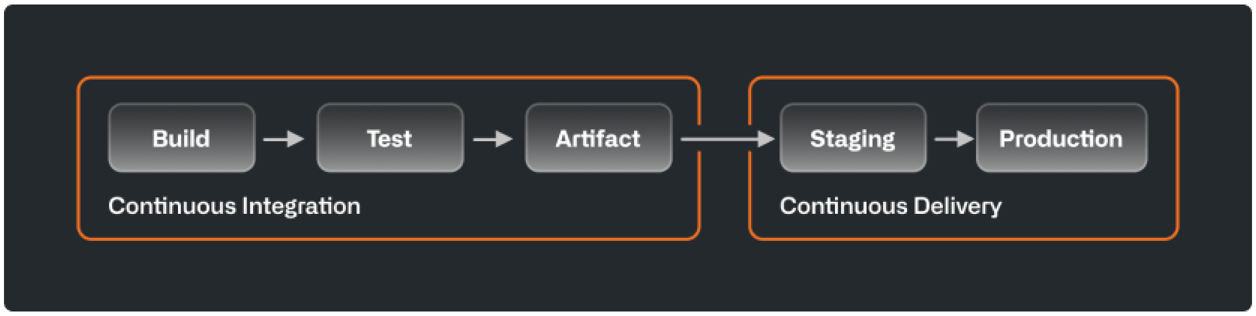
- Understand CI/CD concepts

What CI/CD:

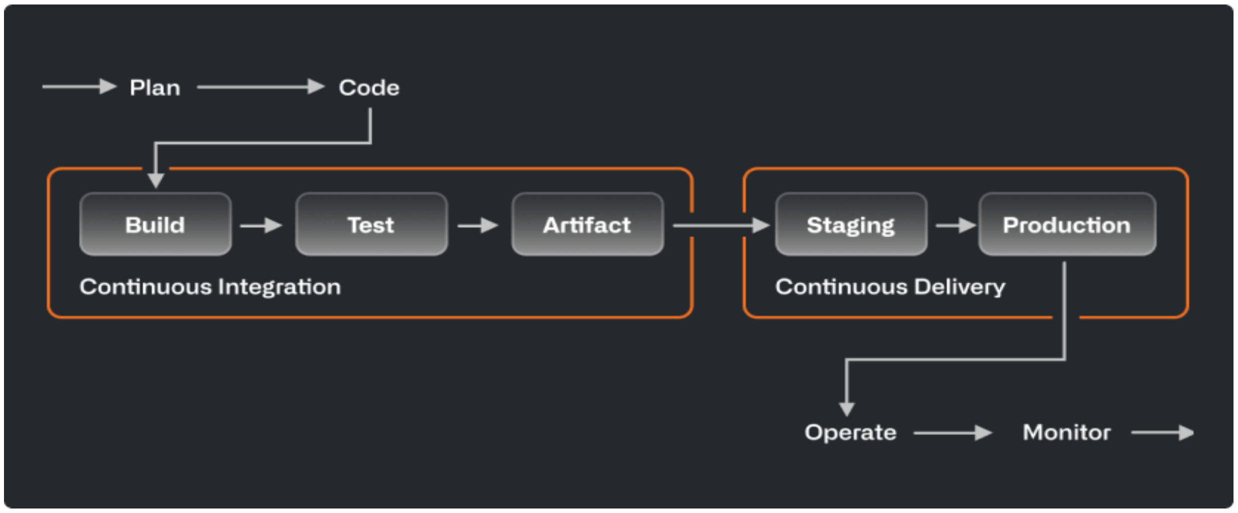
CI/CD automates your builds, testing, and deployment so you can ship code changes faster and more reliably.

Automation is a core principle for achieving DevOps success and CI/CD is a critical component. CI/CD comprises of continuous integration and continuous delivery or continuous deployment. Put together, they form a "CI/CD pipeline"—a series of automated workflows that help DevOps teams cut down on manual tasks:

- **Continuous Integration (CI):** Automates building, testing, and integrating code in a shared repository.
- **Continuous Delivery (CD):** Automatically delivers production-ready code to environments; requires manual approval before release.
- **Continuous Deployment:** Fully automated release to production.



Continuous delivery vs. continuous deployment



Lab # 23

When someone says CI/CD, the "CD" they're referring to is usually continuous *delivery*, not continuous *deployment*. What's the difference? In a CI/CD pipeline that uses continuous delivery, automation pauses when developers push to production. A human—your operations, security, or compliance team—still needs to manually sign off before final release, adding more delays. On the other hand, continuous deployment automates the entire release process. Code changes are deployed to customers as soon as they pass all the required tests.

Continuous deployment is the ultimate example of DevOps automation. That doesn't mean it's the only way to do CI/CD, or the "right" way. Since continuous deployment relies on rigorous testing tools and a mature testing culture, most software teams start with continuous delivery and integrate more automated testing over time.

Why CI/CD?

The short answer: Speed. The State of DevOps report found organizations that have "mastered" CI/CD deploy 208 times more often and have a lead time that is 106 times faster than the rest. While faster development is the most well-known benefit of CI/CD, a continuous integration and continuous delivery pipeline enables much more.

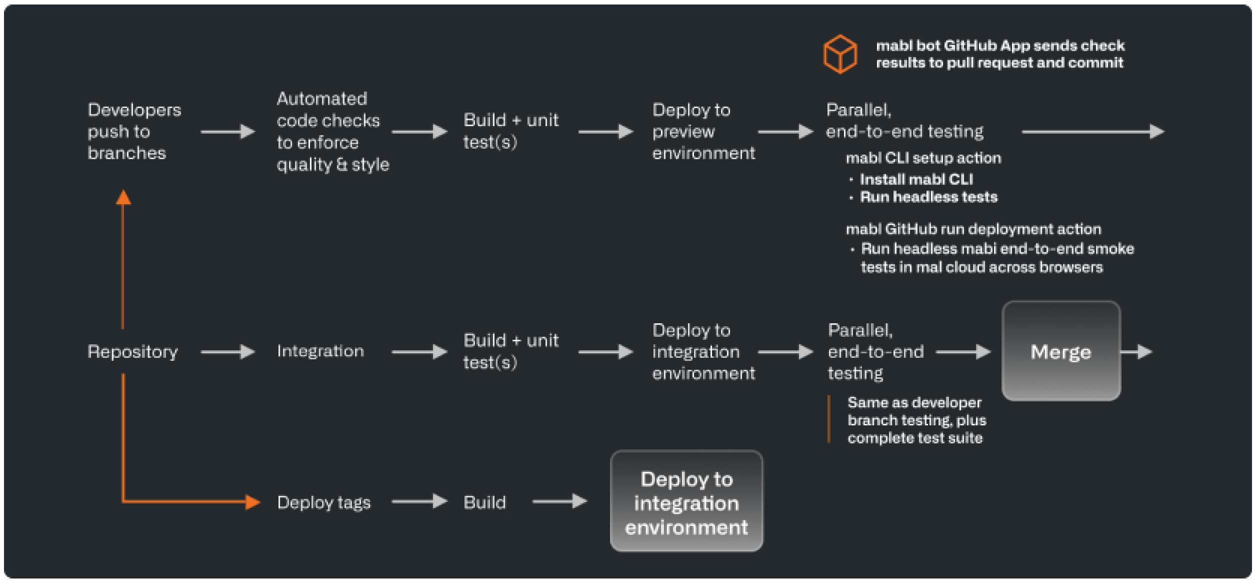
- **Development velocity:** Ongoing feedback allows developers to commit smaller changes more often, versus waiting for one release.
- **Stability and reliability:** Automated, continuous testing ensures that codebases remain stable and release-ready at any time.
- **Business growth:**
  - ✓ Freed up from manual tasks, organizations can focus resources on innovation, customer satisfaction, and paying down technical debt.
  - ✓ The mindset we carry is that we always want to automate ourselves into a better job. We want to make sure that the task we're doing manually today becomes mostly automated.
  - ✓ Andrew Mulholland, Director of Engineering
- **Building your CI/CD toolkit:** Teams make CI/CD part of their development workflow with a combination of automated process, steps, and tools.
- **Version control:** CI begins in shared repositories, where teams collaborate on code using version control systems (VCS) like Git. A VCS tracks code changes, simplifies reversions, and supports config as code for managing testing and infrastructure.
- **Builds:** CI build tools automatically package up files and components into release artifacts and run tests for quality, performance, and other requirements. After clearing required checks, CD tools send builds off to the operations team for further testing and staging.
- **Reviews and approvals:** Treating code review as a best practice improves code quality, encourages collaboration, and helps even the most experienced developers make better commits. In a CI/CD workflow, teams review and approve code or leverage integrated development environments for pair programming.

Lab # 23

- **Environments:** CI/CD tests and deploys code in environments, from where developers build code to where operations teams make applications publicly available. Environments often have their own specific variables and protection rules to meet security and compliance requirements.

Example CI/CD workflow

CI/CD doesn't have to be complicated or mean adding a host of tools on top of your current workflow. At mabl, developers deploy to production about 80 times a week using only two CI/CD integrations: The mabl testing suite and GitHub Actions. Here's how it works.


mabl bot GitHub App sends check results to pull request and commit
Developers push to branches
Automated code checks to enforce quality & style
Build + unit test(s)
Deploy to preview environment
Parallel, end-to-end testing
mabl CLI setup action
• Install mabl CLI
• Run headless tests
mabl GitHub run deployment action
• Run headless mabi end-to-end smoke tests in mal cloud across browsers
Repository
Integration
Build + unit test(s)
Deploy to integration environment
Parallel, end-to-end testing
Merge
Same as developer branch testing, plus complete test suite
Deploy tags
Build
Deploy to integration environment

How CI/ CD:

Let's create a CI/CD pipeline for a simple Flask application using Azure DevOps**.**

Step 1 – Flask Application Setup

1. Create a folder on your system:
   ✓ **flask_ci_cd_lab**

2. Inside the folder, create app.py:

Lab # 23

Program Code

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def index():
   return "Welcome to Flask CI/CD Lab!"
if __name__ == "__main__":
   app.run(debug=True)
```

3. Initialize Git repository
   - ✓ **git init**
   - ✓ **git add .**
   - ✓ **git commit -m "Initial commit for Flask app"**

4. Push to Azure Repos or GitHub.

Step 2 – Azure DevOps Project

1. Log in to Azure DevOps.
2. Create a new **Project** (e.g., FlaskCI_CD).
3. Create a **Repository** or link your existing Git repo.

Step 3 – Create CI/CD Pipeline

YAML Pipeline (Recommended)

1. In Azure DevOps, go to **Pipelines** → **Create Pipeline** → **YAML**.
2. Select your repository.
3. Create a **pipeline YAML** file (e.g., azure-pipelines.yml) in the repo:

Lab # 23

Program Code:

```
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: UsePythonVersion@0
  inputs:
    versionSpec: '3.x'

- script: |
    python -m pip install --upgrade pip
    pip install flask
  displayName: 'Install dependencies'

- script: |
    python -m pytest
  displayName: 'Run Tests'

- script: |
    python app.py
  displayName: 'Run Flask App'
```

Step 4 – Test Pipeline

1. Commit any change to app.py or index.html.
2. Push to repo.
3. Pipeline should trigger automatically:
   - Build runs
   - Tests executed
   - (Optional) Deployment to staging

Step 5 – Verify Deployment

1. If deploying to Azure App Service:
   - Navigate to App Service URL
   - You should see:

Lab # 23

Output:

Welcome to Flask CI/CD Lab!

Task:

Create Your First Flask App's CI/CD Pipeline Using Azure DevOps;

- **Repository URL** (Azure DevOps or GitHub)
- **Screenshot** of:
  - Successful build
  - Successful deployment
  - Flask app running in browser
- **azure-pipelines.yml** (if using YAML pipeline)
- **Short report** (2–3 lines) explaining what CI/CD automated in their pipeline.