# **Introduction to DevOps & CI/CD**

## **What is DevOps?**

DevOps is set of practices that combines software development (Dev) and IT operations (Ops) to shorten the system development life cycle while delivering features, fixes, and updates frequently in close alignment with business objectives.

### **Key Principles of DevOps:**

* Collaboration between development and operation teams
* Automation of processes wherever possible
* Continuous Improvement through feedback loops
* Customer Centric approach to development
* End-to-End responsibility from development to deployment

### **The DevOps Lifecycle**

1. Plan – Define requirements and track progress
2. Code – Software development and review
3. Build – Compile code into executable form
4. Test – Automated testing to verify functionality
5. Release – Prepare for deployment
6. Deploy – Move to Production environment
7. Operate – Monitor and manage the production environment
8. Monitor – Gather performance data and user feedback

### **What is CI/CD?**

CI/CD stands for Continuous Integration and Continuous Delivery/Deployment, which are key practices in DevOps.

### **Continuous Integration (CI)**

* Developers frequently merge code changes into a central repository
* Each merge triggers automated builds and tests
* Early detection of Integration issues
* Requires Comprehensive test suites

### **Continuous Delivery (CD)**

* Ensures code is always in a deployable state
* Automated deployment to staging environments
* Manual approval for production deployment

### **Benefits of DevOps and CI/CD**

1. **Faster time to market –** More frequent releases
2. **Improved quality –** Early bug detection
3. **Reduced Risk –** Smaller changes are easier to troubleshoot
4. **Greater efficiency –** Automation reduces manual work
5. **Better collaboration –** Shared responsibility across teams
6. **Increased reliability –** Consistent deployment processes

### **Common DevOps Tools**

* **Version Control:** Git, GitHub, GitLab, Bitbucket
* **CI/CD Servers:** Jenkins, CicleCI, Travis CI, GitHub Actions
* **Configuration Management:** Ansible, Chef, Puppet
* **Containerization:** Docker, Kubernetes
* **Monitoring:** Prometheus, Grafana, ELK Stack
* **Cloud Platforms:** AWS, Azure, Google Cloud

## **Traditional vs. DevOps Workflow: Key Differences**

The main difference between **Traditional (Waterfall/Siloed) workflows** and **DevOps workflows** lies in their approach to software development, collaboration, and deployment. Below is a detailed comparison:

**1. Development Approach**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Traditional Workflow** | **DevOps Workflow** |
| **Methodology** | Follows a **linear (Waterfall)** approach (Requirements -> Design -> Development -> Testing -> Deployment) | Follows an Agile & Iterative approach with continuous integration and delivery. |
| **Team Structure** | **Siloed teams** (Dev, QA, Ops work separately) | Cross-functional teams (Dev + Ops Collaborate closely) |
| **Feedback Loop** | Long feedback cycles (weeks/months) | Short feedback loops (minutes/hours via CI/CD) |

**2. Release & Deployment**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Traditional Workflow** | **DevOps Workflow** |
| **Release Frequency** | Infrequent (months/Years) | **Frequent releases** (daily/hourly) |
| **Deployment** | Manual, error-prone and slow | **Automated** (CI/CD pipeline handle builds, tests, and deployments) |
| **Risk of Failures** | High (big releases = big failures) | **Low** (small, incremental changes) |

**3. Testing & Quality Assurance**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Traditional Workflow** | **DevOps Workflow** |
| **Testing Phase** | Separate phase after development (slow feedback) | Continuous Testing (automated tests |
| **Bug Detection** | Bugs found late (costly to fix) | Bugs caught early (shift-left testing) |

**4. Infrastructure & Operations**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Traditional Workflow** | **DevOps Workflow** |
| Infrastructure | Manual setup, static servers | Infrastructure as Code (IaC) (Terraform, Ansible) |
| Scaling | Difficult (requires manual intervention) | Auto-scaling (Cloud, Kubernetes) |
| Monitoring | Reactive (issues detected after failure) | Proactive monitoring (logs, metrics, alerts |

**5. Culture & Collaboration**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Traditional Workflow** | **DevOps Workflow** |
| Team mindset | “Throw it over the wall” (Dev -> QA -> Ops) | Shared responsibility (Dev + Ops work together) |
| Blame Game | Common (Ops blames Dev for bugs, Dev blames Ops for environment issues) | No blame culture (focus on automation & collaboration) |

# **Version Control with Git & GitHub**

## **Introduction to Version Control**

Version control is a system that records changes to files or sets of files over time, allowing multiple users to collaborate, track history, and revert to previous versions when needed. It is an essential tool in software development, content creation, and any project involving iterative changes. Version control systems (VCS) address the challenges of managing evolving files by providing structure, accountability, and collaboration capabilities.

**Why Use Version Control?**

* **Track Changes**: Every modification to a file is recorded with details about who made the change, when, and why, enabling a clear history of the project.
* **Collaboration**: Multiple team members can work on the same project simultaneously without overwriting each other’s work.
* **Revert Mistakes**: If a change introduces errors, you can revert to a previous version, ensuring mistakes are not permanent.
* **Branching and Experimentation**: Developers can create separate branches to test new features or ideas without affecting the main project.
* **Backup and Recovery**: Version control acts as a backup, storing all versions of files in a repository, reducing the risk of data loss.
* **Auditability**: Changes are documented, making it easier to review code, debug issues, or comply with regulatory requirements.

**Types of Version Control Systems**

1. **Local Version Control**: Simple systems that track changes on a single machine (e.g., copying files into dated folders or using tools like RCS).
2. **Centralized Version Control**: A single server stores the repository, and users check out files to make changes (e.g., SVN, CVS). However, it has a single point of failure.
3. **Distributed Version Control**: Every user has a complete copy of the repository, including its history, enabling offline work and greater resilience (e.g., Git, Mercurial).

Version control is critical for modern workflows, especially in software development, where teams need to manage complex codebases efficiently.

**What is Git & Why Use It?**

**Git** is a distributed version control system designed to handle everything from small to large projects with speed and efficiency. Created by Linus Torvalds in 2005 for Linux kernel development, Git has become the de facto standard for version control due to its flexibility and performance.

**Key Features of Git**

* **Distributed Architecture**: Every user has a full copy of the repository, including its history, allowing offline work and reducing dependency on a central server.
* **Speed**: Git is optimized for fast operations, such as committing changes, branching, and merging, even in large repositories.
* **Branching and Merging**: Git’s lightweight branching model allows users to create, merge, and manage multiple branches easily, facilitating parallel development.
* **Data Integrity**: Git uses SHA-1 hashes to ensure the integrity of commits, making it nearly impossible to lose or corrupt data.
* **Flexibility**: Git supports various workflows (e.g., Gitflow, GitHub Flow) and integrates with platforms like GitHub, GitLab, and Bitbucket.
* **Open Source**: Git is free, open-source, and maintained by a large community, ensuring continuous improvement and support.

**Why Use Git?**

* **Collaboration**: Git enables seamless collaboration by allowing multiple developers to work on the same project without conflicts, using features like branches and merges.
* **Scalability**: It handles projects of any size, from personal scripts to massive open-source repositories like the Linux kernel.
* **Community and Ecosystem**: Git’s widespread adoption means extensive documentation, tutorials, and integrations with tools like GitHub, CI/CD systems, and IDEs.
* **Error Recovery**: Commands like reset, revert, and reflog allow users to undo mistakes or recover lost work.
* **Open-Source Contributions**: Git is the backbone of platforms like GitHub, making it essential for contributing to open-source projects.

Git’s combination of power, speed, and flexibility makes it indispensable for developers, data scientists, and anyone managing iterative projects.

## Difference Between Git and GitHub

While Git and GitHub are closely related, they serve distinct purposes:

**Git**

* **Definition**: Git is a distributed version control system, a command-line tool that runs locally on your computer to track changes in files and manage repositories.
* **Functionality**: It handles versioning, branching, merging, and committing without requiring an internet connection.
* **Storage**: Repositories are stored locally, and you can push them to remote servers if needed.
* **Usage**: Git is platform-agnostic and can be used with any hosting service or none at all.
* **Example Commands**: git init, git commit, git branch, git merge.

**GitHub**

* **Definition**: GitHub is a web-based platform for hosting Git repositories, collaborating on projects, and adding features like pull requests, issues, and CI/CD.
* **Functionality**: It provides a user-friendly interface for managing Git repositories, enabling collaboration through features like pull requests, code reviews, and project boards.
* **Storage**: Repositories are stored in the cloud, accessible via Git commands or GitHub’s web interface.
* **Usage**: GitHub is a specific service that relies on Git but adds social and collaborative tools, such as forking, starring, and GitHub Actions.
* **Example Features**: Creating pull requests, managing issues, setting up GitHub Pages, automating workflows with GitHub Actions.

**Key Differences**

|  |  |  |
| --- | --- | --- |
| Aspect | Git | GitHub |
| **Nature** | Command-line tool | Web-based platform |
| **Purpose** | Version control | Hosting and collaboration |
| **Dependency** | Standalone | Requires Git |
| **Access** | Local (offline) | Cloud-based (online) |
| **Features** | Core VCS operations | Pull requests, issues, Actions |
| **Cost** | Free (open-source) | Free tier + paid plans |

In summary, Git is the underlying technology for version control, while GitHub is a platform that enhances Git with collaboration and hosting features. You can use Git without GitHub, but GitHub relies on Git.

## **Installing Git (Windows, macOS, Linux)**

Installing Git is straightforward, and the process varies slightly depending on the operating system. Below are detailed instructions for each platform.

### **Windows**

1. **Download Git**:
   * Visit the official Git website: https://git-scm.com/download/win.
   * The download should start automatically for the latest version (e.g., Git 2.43.0 as of May 2025).
2. **Run the Installer**:
   * Open the downloaded .exe file (e.g., Git-2.43.0-64-bit.exe).
   * Follow the setup wizard:
     + **Components**: Select “Git Bash Here” and “Git GUI Here” for convenience.
     + **Default Editor**: Choose an editor like Notepad++ or VS Code (or keep the default, Vim).
     + **Adjusting PATH**: Select “Git from the command line and also from 3rd-party software” for broad compatibility.
     + **Line Endings**: Choose “Checkout Windows-style, commit Unix-style line endings” for cross-platform compatibility.
     + Keep other defaults unless you have specific needs.
3. **Verify Installation**:
   * Open Git Bash (or Command Prompt) and run:

git --version

* + Expected output: git version 2.43.0 (or the installed version).

1. **Optional**: Install Git Credential Manager for easier authentication with remote repositories.

### **macOS**

1. **Using Homebrew (Recommended)**:
   * Install Homebrew if not already installed:

/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

* + Install Git:

brew install git

1. **Alternative: Download Installer**:
   * Visit https://git-scm.com/download/mac.
   * Download the installer and follow the setup wizard.
2. **Verify Installation**:
   * Open Terminal and run:

git --version

* + Expected output: git version 2.43.0 (or the installed version).

1. **Note**: macOS includes an older version of Git by default. Using Homebrew ensures you get the latest version.

### **Linux**

1. **Ubuntu/Debian**:
   * Update package lists:

sudo apt update

* + Install Git:

sudo apt install git

1. **Fedora**:
   * Install Git:

sudo dnf install git

1. **Arch Linux**:
   * Install Git:

sudo pacman -S git

1. **Verify Installation**:
   * Open a terminal and run:

git --version

* + Expected output: git version 2.43.0 (or the installed version).

1. **Note**: Use your distribution’s package manager for the latest Git version.

**Post-Installation**

* Ensure Git is accessible from your preferred terminal (e.g., Bash, PowerShell, or Zsh).
* Update Git periodically to get the latest features and security patches.

## **Basic Git Configuration (git config)**

After installing Git, you need to configure it with your identity and preferences. The git config command sets up user-specific settings stored in configuration files.

**Key Configuration Steps**

### **Set User Identity:**

* + Git associates your commits with a name and email address. Run:
  + git config --global user.name "Your Name"

git config --global user.email "your.email@example.com"

* + Example:

git config --global user.name "Jane Doe"

git config --global user.email "jane.doe@example.com"

* + The --global flag applies settings to all repositories on your machine. Omit it for repository-specific settings.

### **Set Default Editor:**

* + Git uses a text editor for commit messages and interactive commands (e.g., rebase). Set your preferred editor:

git config --global core.editor "code --wait"

* + Common editors:
    - VS Code: code --wait
    - Vim: vim
    - Nano: nano
    - Emacs: emacs
  + If not set, Git defaults to Vim.

### **Set Default Branch Name:**

* + Git uses main or master as the default branch name. To standardize:

git config --global init.defaultBranch main

* + This ensures new repositories use main as the default branch.

### **Enable Color Output:**

* + Improve readability of Git output:

git config --global color.ui auto

* + This enables colored output for commands like git status and git diff.

### **Verify Configuration:**

* + List all global settings:

git config --global --list

* + Example output:
  + user.name=Jane Doe
  + user.email=jane.doe@example.com
  + core.editor=code --wait
  + init.defaultBranch=main

color.ui=auto

### **Configuration Files**

* **Global**: Stored in ~/.gitconfig (or %USERPROFILE%\.gitconfig on Windows).
* **Repository-Specific**: Stored in .git/config within a repository.
* **System-Wide**: Stored in /etc/gitconfig (rarely used).

### **Additional Configurations (Optional)**

* **Aliases**: Simplify commands with shortcuts:
* git config --global alias.co checkout
* git config --global alias.br branch
* git config --global alias.st status
  + Example: git co now runs git checkout.
* **Credential Helper**: Store credentials for remote repositories:

git config --global credential.helper cache

* + On Windows, use credential.helper=manager for Git Credential Manager.

**Best Practices**

* Use a consistent email address (e.g., the one linked to your GitHub account).
* Keep configurations minimal to avoid conflicts.
* Back up ~/.gitconfig when switching machines.

This introduction and detailed explanation provide a solid foundation for understanding version control, Git, and GitHub, along with practical steps to get started. You’re now ready to initialize repositories, track changes, and collaborate on projects!

## **Git Commands & Workflow**

Git is a distributed version control system that helps changes in files and collaborate on projects. Below, we will break down the Git workflow and demonstrate key commands with examples.

### **Git Workflow Overview**

Git follows a three-stage workflow:

1. Working Directory – Where you modify files
2. Staging Area (Index) – Where you prepare changes to commit
3. Repository (History) – Where committed changes are permanently stored

Working Directory -> ‘git add’ -> Staging Area -> ‘git commit’ -> Repository

### **Basic Git Commands with Examples**

1. Creating & Cloning a Repository

*# Initialize a new Git repo*

git init

*# Clone an existing repo from GitHub*

git clone <https://github.com/user/repo.git>

1. Tracking Changes

*# Check file status*

git status

*# Add a file to staging*

git add filename.txt

*# Add all changes to staging*

git add .

*# Commit changes with a message*

git commit -m "Added new feature"

1. Viewing History

*# Show commit history*

git log

*# Show compact log*

git log --oneline

*# See changes in a commit*

git show <commit-hash>

### **Basic Branching and Merge Commands with Examples**

1. Creating & Switching Branches

*# List all branches*

git branch

*# Create a new branch*

git branch feature-login

*# Switch to a branch*

git checkout feature-login

*# OR (newer Git versions)*

git switch feature-login

1. Merging Branches

*# Switch to `main` branch*

git switch main

*# Merge `feature-login` into `main`*

git merge feature-login

1. Handling Merge Conflicts

If two branches modify the same file, Git may show:

CONFLICT (content): Merge conflict in file.txt

Open the file, resolve conflicts (remove <<<<<<<, =======, >>>>>>> ), then:

git add file.txt

git commit -m "Resolved merge conflict"

### **Remote Repositories (GitHub)**

Pushing and Pulling Changes

Forking & Pull Requests (PRs)

### **Undoing Changes**

|  |  |  |
| --- | --- | --- |
| Command | Use Case | Example |
| git restore file.txt | Discard unstaged changes | git restore script.js |
| git reset –soft HEAD~1 | Undo last commit (keep changes staged) | git reset –soft HEAD~1 |
| git reset –hard HEAD~1 | DANGER! Permanently discard last commit | git reset –hard HEAD~1 |
| git revert <commit-hash> | Safely undo a commit (creates a new commit) | git revert abc1234 |
| git stash | Temporarily save uncommitted changes | Git stash pop (to restore) |

### **Advanced Git Commands**

#### **Rebasing (Cleaner History)**

git switch feature

git rebase main # Reapply commits on top of `main`

#### **Cherry Picking (Copy a Commit)**

git cherry-pick abc1234 # Apply commit `abc1234` to current branch

#### **Reflog (Recover Lost Commits)**

git reflog # View all Git actions

git checkout abc1234 # Recover a lost commit

### **Git Workflow Example**

Scenario: Fix a bug in a project

1. Pull latest changes

git pull origin main

1. Create a new branch

git switch -c fix-bug

1. Make Changes & Commit

git add .

git commit -m "Fixed login bug"

1. Push to GitHub & Open PR

git push origin fix-bug

1. After PR is merged

git switch main

git pull origin main

git branch -d fix-bug # Delete local branch

# **Introduction to CI/CD Pipelines**

Continuous Integration (CI) and Continuous Deployment are essential practices in modern software development. They help teams automate and streamline the process of integrating code changes, testing, and deploying applications. CI/CD pipelines enable developers to deliver high-quality software efficiently and reduce manual intervention in the deployment process.

**What is CI/CD?**

CI/CD Stands for:

* **Continuous Integration (CI):** The process of automatically integrating code changes into a shared repository multiple times a day. It involves automated testing to ensure new code does not introduce errors.
* **Continuous Deployment (CD):** The practice of automatically deploying code changes to production after passing the necessary tests. This ensures that software is always in a deployable state.

**Key Components of a CI/CD Pipeline**

A typical CI/CD pipeline consists of several stages:

1. **Source Code Management:** Developers commit their code to a version control system like Git.
2. **Build stage**: The Source code is compiled, dependencies are installed, and the application is packaged
3. **Testing stage:**
   * **Unit testing:** Validates individual components of the code
   * **Integration Testing:** Ensures that different modules work together
   * **Functional Testing:** Confirms that the application meets business requirements
4. **Deployment Stage**
   * **Staging Environment:** A Pre-production setup where tests are run before final deployment
   * **Production Deployment:** The application is released for users
5. **Monitoring & Feedback:** Logs and metrics are collected to monitor application performance and detect issues.

**Benefits of CI/CD pipelines**

**Speed & Efficiency:** Automates the development cycle, reducing manual effort.

**Reliability:** Ensure consistent deployments and minimized errors

**Scalability:** Enables teams to handle large codebases with multiple contributors

**Improved Collaboration:** Developers work in sync, reducing integration conflicts.