

---

# 1 Introduction to Git and GitHub

**Git** and **GitHub** are two closely related but distinct tools used in software development. While Git focuses on *version control* and tracking changes to source code, GitHub provides a *cloud-based platform* for hosting Git repositories and enabling collaboration among developers. Together, they form the backbone of modern software workflows, allowing individuals and teams to manage projects efficiently, maintain history, and contribute from anywhere.

## Interesting Fact

Git stores data as snapshots of your entire project (not just file diffs). When nothing changes, Git simply points to the previous snapshot—one reason it's fast.

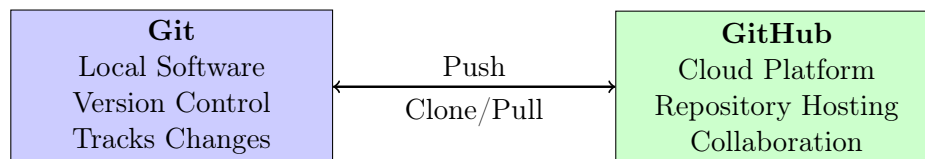


Figure 1: Git vs GitHub Relationship

## 1.1 What is Git?

**Git** is a *distributed version control system* that allows developers to:

- Track changes in files over time.
- Maintain a full history of a project locally.
- Revert to previous versions when needed.
- Work offline and later synchronize with others.

It was created by *Linus Torvalds* in 2005 to manage the development of the Linux kernel and has since become the most widely used version control system.

## Git Definition

Git runs on your local computer and is designed to be **fast, reliable, and scalable**. It supports branching, merging, and collaboration in both small and large projects.

## 1.2 What is GitHub?

**GitHub** is a *cloud-based platform* that provides hosting services for Git repositories. It acts as a central hub where developers can:

- Store their repositories online as a backup.
- Collaborate with others through pull requests and code reviews.
- Use project management tools like issues, discussions, and wikis.
- Integrate workflows with automation and continuous integration (CI/CD).

---

Founded in 2008 and later acquired by Microsoft, GitHub has grown into one of the largest development platforms, connecting millions of developers worldwide.

#### GitHub Definition

GitHub extends Git by offering a **social coding environment**. It makes open-source collaboration, code sharing, and teamwork seamless.

### 1.3 Key Difference

Although closely connected, the distinction is simple:

- **Git** = The tool (software) installed on your computer to track and manage changes.
- **GitHub** = The service (website/platform) that hosts Git repositories online for collaboration.

In short: *Git is the engine, GitHub is the garage.*

---

## 2 Installation and Setup

Before using Git and GitHub, you need to set up the essential tools on your system. The installation process includes installing Git itself, setting up a text editor or IDE, and verifying that everything works correctly. This ensures that you can start version controlling your projects smoothly.

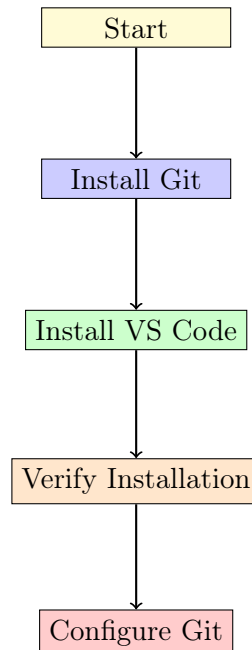


Figure 2: Installation Process Flow

### Did You Know?

On Windows, installing “Git for Windows” also gives you *Git Bash*, a Unix-like shell that makes many tutorials and commands consistent across platforms.

### 2.1 Installing Git

**Git** is required on your local machine to begin version controlling files. Installation steps vary depending on your operating system:

- **Windows:** Download and install Git for Windows from the official site. It comes with *Git Bash*, a terminal to run Git commands.
- **Mac:** Git is often pre-installed. You can check with: `git -version` If not installed, you can install it via *Homebrew* using: `brew install git`
- **Linux:** Use your distribution’s package manager. For example: `sudo apt-get install git` (Debian/Ubuntu) `sudo yum install git` (Fedora/RedHat)

**Verify Installation:** After installation, run:

```
git --version
```

This should display the installed version of Git.

### Additional Resource

[GIT INSTALLATION \(PDF\)](#)

---

## 2.2 Installing VS Code

**Visual Studio Code (VS Code)** is a free, lightweight, and powerful code editor from Microsoft. It integrates well with Git and GitHub.

1. Visit the official Visual Studio Code website.
2. Download the installer for your operating system (*Windows, Mac, or Linux*).
3. Run the installer and follow the setup wizard instructions.
4. Once installed, open VS Code and explore built-in features like syntax highlighting, extensions, and the integrated terminal.

### Additional Resource

**VS CODE INSTALLATION** (PDF)

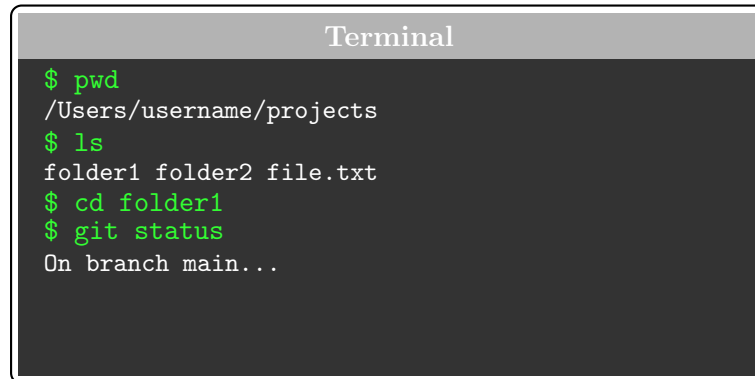
### Tip

Installing the **GitLens** extension in VS Code is highly recommended. It enhances Git integration by showing commit history, blame annotations, and repository insights directly in the editor.

---

## 3 Essential Terminal Commands

The terminal (or command line) is the main way you interact with Git. It allows you to navigate through directories, run Git commands, and manage your projects efficiently. Learning some basic terminal commands is essential before diving into Git workflows.



```
Terminal
$ pwd
/Users/username/projects
$ ls
folder1 folder2 file.txt
$ cd folder1
$ git status
On branch main...
```

Figure 3: Terminal Command Examples

### 3.1 Basic Navigation

These commands help you move around your computer's directory structure using the terminal:

Navigation Commands	
<code>pwd</code>	# Show current directory
<code>ls</code>	# List files and folders
<code>ls -a</code>	# List all files including hidden
<code>clear</code>	# Clear terminal (Windows: <code>cls</code> )
<code>cd [directory_name]</code>	# Change directory
<code>cd ..</code>	# Go back to parent directory
<code>mkdir [folder_name]</code>	# Create a new folder

#### Shortcut Spotlight

Tab auto-completes paths; **Ctrl+C** cancels the current command; **Up/Down** cycles through command history. Small habits, big speed.

#### Tips:

- Use **Tab** for auto-completion of folder or file names.
- Use the **Up/Down arrow** keys to quickly access previously used commands.

### 3.2 VS Code Terminal

Visual Studio Code has a built-in terminal that lets you run Git commands without leaving the editor.

- **Mac:** Press **Control + ~** (tilde key).
- **Windows:** Press **Control + `** (single quote) or go to **Menu → View → Terminal**.

You can open multiple terminal tabs, and even choose the shell you prefer:

- 
- **Git Bash** (recommended for Windows users).
  - **Command Prompt (cmd)** or **PowerShell**.
  - **Default terminal** on Mac/Linux.

## Terminal Power Moves (Quick Guide)

The terminal shines when you combine tiny commands into repeatable habits. These moves save seconds that add up across a project.

- **Glance at where you are:** `pwd` prints your current path; keep your prompt short and your path long in muscle memory.
- **List smarter:** `ls -lah` shows human-readable sizes; `ls -lt` sorts by time so the latest files are at the top.
- **Navigate fast:** `cd -` toggles between the last two folders; `cd ..` goes up; `cd ../..` jumps two levels.
- **Search within files:** `grep -n "text" file` shows matching line numbers; add `-R` to search recursively.
- **Create and open quickly:** `touch notes.md` then `code notes.md` (or your editor) to start writing.
- **Chain commands:** Use `&&` so the next command only runs if the previous succeeded: `npm test && git commit -m "green"`.
- **Pipe output:** `git log --oneline | head -n 10` shows just the most recent 10 commits.
- **History wins:** Press `Ctrl+R` and type a fragment to reverse-search your command history; hit `Enter` to run.
- **Cancel/quit:** `Ctrl+C` cancels the current program; `q` usually quits pagers like `less`.

### Handy One-Liners

```
# Find the biggest files (top 10) in the current folder
du -ah . | sort -hr | head -n 10

# Count lines of code by extension (e.g., .js)
find . -name "*.js" -print0 | xargs -0 wc -l | sort -nr | head

# Save command output to a file and view it
git status > status.txt && code status.txt
```

### Quick Recap

Know your way around `pwd`, `ls`, and `cd`; use reverse search (`Ctrl+R`); and chain with `&&` to build reliable, repeatable workflows right from the terminal.

---

## 4 Git Configuration

Before using Git for the first time, you need to configure your identity and preferences. This step ensures that every commit you make will be associated with your name and email address. Without configuration, Git will not know who you are, and collaboration with others becomes difficult.

Git configuration is done using the `git config` command. There are three levels of configuration:

- **System level:** Applies to all users on the computer.
- **Global level:** Applies to your user account across all repositories (most common).
- **Local level:** Applies only to a specific repository.

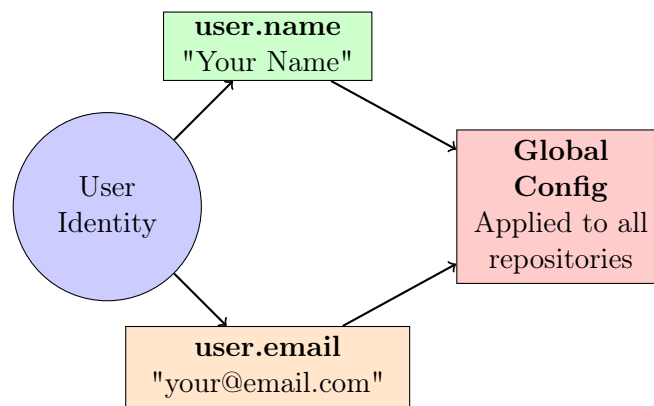


Figure 4: Git Configuration Setup

### 4.1 Setting Your Identity

The first step is to set your name and email. These will appear in every commit you make.

#### Git Configuration Commands

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

#### Quick Trick

Edit your global config in your default editor with: `git config -global -edit`. Handy for tweaking multiple settings at once.

#### Explanation:

- The flag `-global` means the setting applies to all repositories on your computer.
- If you omit `-global`, the setting applies only to the current repository (local config).
- You can override global settings with repository-specific local settings if needed.

---

## 4.2 Verifying Your Configuration

To check your configuration, run:

Check Settings

```
git config --list
```

This will show all your Git settings, including `user.name` and `user.email`.

You can also check individual values:

```
git config user.name
git config user.email
```

## 4.3 Other Useful Configurations

Besides name and email, Git allows you to configure other settings:

- **Default editor:** Set the text editor Git should use (for commit messages, merge messages, etc.):

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

(`code -wait` sets VS Code as the default editor.)

- **Default branch name:** Change the initial branch from `master` to `main`:

```
git config --global init.defaultBranch main
```

- **Colored output:** Make Git show colored output in the terminal for better readability:

```
git config --global color.ui auto
```

## 4.4 Where Configurations Are Stored

- **System:** Stored in `/etc/gitconfig`.
- **Global:** Stored in `~/.gitconfig` (user's home directory).
- **Local:** Stored in `.git/config` inside each repository.

Git reads settings in the following order of priority: **Local** → **Global** → **System**. This means local settings override global ones, and global settings override system ones.

## 4.5 Best Practices

- Always set your global identity (`user.name` and `user.email`) immediately after installing Git.
- Use your real name and a valid email address for collaboration (GitHub will link commits to your account).
- Configure a default editor you are comfortable with (VS Code, Vim, Nano, etc.).
- Use descriptive commit messages – Git will open your editor if you don't provide one inline.



---

## 5 GitHub Repository Management

A **repository (repo)** is the core element of Git and GitHub. It acts as a storage space where your project files, history, and configuration are kept. Repositories can be private (visible only to you and selected collaborators) or public (open to everyone).

On GitHub, repositories allow you to:

- Safely back up your code in the cloud.
- Collaborate with teammates across the globe.
- Share your projects publicly with the open-source community.
- Track the complete history of your work.

Managing repositories efficiently is essential for both individual projects and team workflows.

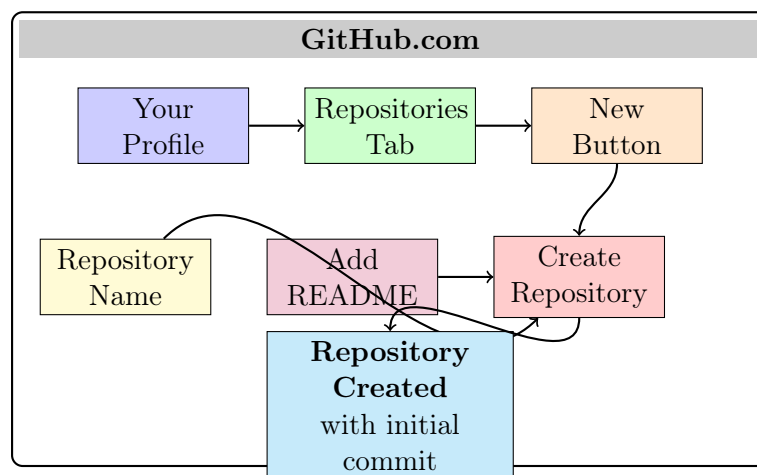


Figure 5: GitHub Repository Creation

### Nice to Know

You can rename a repository on GitHub; existing clones keep working because GitHub provides transparent redirects for the old remote URL.

### 5.1 Creating a Repository on GitHub

To start a new project on GitHub, you first need to create a repository. Follow these steps:

1. **Log in** to your GitHub account at [www.github.com](https://www.github.com).
2. Go to your profile and click on the **Repositories** tab.
3. Click on the green **New** button to create a new repository.
4. Enter a meaningful **repository name**. Use lowercase letters and hyphens for readability (e.g., `my-first-project`).
5. Add an optional description to explain what your project does.
6. Choose whether your repository should be **Public** or **Private**.
7. Initialize with a **README** file to introduce your project.

- 
8. Optionally add a **.gitignore** file to exclude unnecessary files (e.g., system files, IDE settings).
  9. Select a **license** if you want others to know how they can use your code.
  10. Finally, click **Create repository**.

#### Additional Resource

**GITHUB SETUP** (PDF)

#### Tip

Always initialize your repository with a README file. It gives visitors a clear idea of the project's purpose and makes the repo more professional.

After creation, GitHub will show you options to:

- Clone the repository to your local computer.
- Upload files directly through the web interface.
- Or initialize a new local repository and push it to GitHub.

## 5.2 Repository Features

Repositories are much more than storage spaces. They come with several features that make development efficient and collaborative.

Key features include:

- **Commits:** Every change you make in a repository is recorded as a commit. Commits act as checkpoints in your project's history and include a descriptive message.
- **Commit History:** A timeline view of all the commits, showing what changed, who made the change, and when. This provides accountability and traceability.
- **Branches:** Branches allow you to work on new features or bug fixes without disturbing the main project. The default branch is usually called **main**.
- **Pull Requests (PRs):** A pull request is a way to propose changes. It allows others to review your code, suggest improvements, and approve merging into the main branch.
- **Issues:** A lightweight tracking system for bugs, enhancements, or questions. Each issue can be assigned to a team member, labeled, and closed when resolved.
- **Wiki:** A space for documentation within your repository. Useful for explaining workflows, setups, or technical guides.

#### Best Practices

- Use descriptive repository names and commit messages.
- Regularly push changes to keep your work backed up.
- Create feature branches instead of working directly on the main branch.
- Review and discuss code changes via pull requests.
- Use issues and labels to organize project tasks.

---

In short, GitHub repositories are not just file containers — they are powerful tools for collaboration, project management, and automation. By understanding how to create and use them effectively, you can work like a professional developer and contribute seamlessly to both private projects and large open-source communities.

## Repository Hygiene Checklist

Keep your repo friendly for you and future contributors.

- **README first:** What it is, how to run it, how to test it, how to contribute.
- **.gitignore:** Exclude OS files, build artifacts, logs, and editor caches.
- **License:** Choose one early (MIT/Apache-2.0) so others know how they can use your code.
- **Branches:** Protect `main`; develop in short-lived feature branches; merge via PRs.
- **Labels:** Create a small, meaningful set (e.g., `bug`, `feature`, `good first issue`, `help wanted`).
- **Security:** Enable Dependabot (or similar) to receive dependency alerts and PRs.

### README Skeleton

```
# Project Name
One short sentence describing the problem this solves.

## Quick Start
- Prereqs: Node 20, Docker, etc.
- Install: npm install
- Run: npm start
- Test: npm test

## Usage
Example commands / API calls / screenshots.

## Contributing
How to file issues, create branches, write commits, and open PRs.

## License
MIT (or your chosen license)
```

### Issue Labels You Should Start With

**Type:** bug, feature, docs    **Size:** small, medium, large    **State:** triage, in progress, blocked    **Good for newcomers:** good first issue, help wanted

### Quick Recap

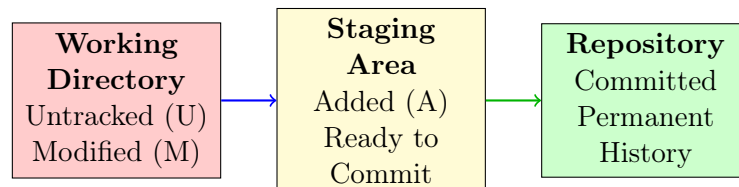
A tidy repo has a clear README, a sensible `.gitignore`, protected `main`, CI for tests, and labels that steer work. Ship small PRs; review quickly.

---

## 6 Git Workflow and File States

Understanding the Git workflow is essential to using Git effectively. Git follows a unique process that separates changes into different areas before they become part of the project's permanent history. This ensures that you have complete control over what is included in your repository at each step.

At the core of Git is a **two-step process**: 1) adding changes to a staging area, and 2) committing them to the repository. Every file in a Git project can be in one of several states depending on where it is in this workflow.



Like engagement → marriage

Figure 6: Git Two-Step Workflow

### 6.1 The Two-Step Process

In Git, changes are not immediately saved to the repository. Instead, they pass through two stages:

1. **Staging Area (Index):** When you make changes to files, they remain in the working directory. To prepare them for a commit, you add them to the staging area using:

```
git add filename
git add . # adds all changes
```

The staging area acts like a “shopping cart” – you decide what changes should be included in the next commit.

2. **Commit:** Once changes are staged, you can save them permanently to the repository's history with a commit:

```
git commit -m "Your commit message here"
```

Each commit stores:

- The exact changes made.
- The author's name and email.
- A timestamp.
- A descriptive commit message.

#### Pro Move

Made a typo in your last message? Fix it with `git commit -amend -m "New message"` (before pushing).

**Analogy:** Think of it like *engagement and marriage*.

- **Staging** = Engagement (changes are prepared but not final).

- **Commit** = Marriage (changes are permanently recorded).

### Best Practice

Always write meaningful commit messages. Instead of “fix” or “update”, describe the actual change, e.g.: `git commit -m "Added login validation to user form"`.

## 6.2 File States in Git

At any point, each file in your project falls into one of several states:

- **Untracked (U):** The file is new and Git is not tracking it yet. Example: you just created `notes.txt`.
- **Modified (M):** The file has been changed since the last commit but has not been staged yet.
- **Staged (A):** The file has been marked with `git add` and is ready to be committed.
- **Committed:** The file’s changes are permanently stored in the repository’s history.

You can check the status of your files with:

```
git status
```

This command shows which files are untracked, modified, or staged.

### Tip

A good workflow is:

1. Edit files in your project.
2. Run `git status` to see changes.
3. Stage files using `git add`.
4. Commit changes with a clear message.

Following this cycle ensures your history is clean and easy to understand.

### Example Workflow:

```
# Create a new file
echo "Hello Git" > hello.txt

# Check status
git status

# Stage the new file
git add hello.txt

# Commit the change
git commit -m "Added hello.txt with greeting message"
```

By repeating this process, you build a structured and traceable history of your project. Over time, the repository becomes a detailed record of how your project evolved.

---

## 7 Essential Git Commands

Git offers a wide range of commands, but as a beginner you only need a core set to manage your workflow. These essential commands allow you to create repositories, check file status, stage changes, and commit them to history.

Mastering these commands forms the foundation of working with Git. Later, you will expand to advanced commands like branching, merging, and remote operations, but for now let's focus on the basics.

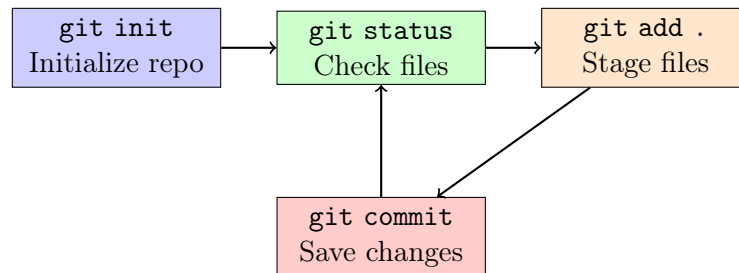


Figure 7: Git Command Flow

### 7.1 Repository Operations

When starting a new project or working with an existing one, you need to create or copy a repository:

#### Repository Commands

```
git init                # Initialize new Git repository
git clone [repository_url] # Copy repository from GitHub
git status              # Check current status of files
```

#### Handy View

Try `git log -oneline -graph -decorate -all` to see a compact, branch-aware history graph in your terminal.

#### Explanation:

- `git init` – Creates a new local repository in the current folder. It adds a hidden `.git` directory to track changes.
- `git clone` – Downloads a remote repository from GitHub (or another server) to your computer.
- `git status` – Shows what files are untracked, modified, staged, or committed.

#### Best Practice

Run `git status` frequently. It helps you avoid mistakes and keeps you aware of which files are staged and which are not.

### 7.2 Staging and Committing

Once a repository exists, the main cycle of work involves adding changes and committing them:

### Staging and Commit Commands

```
git add [filename]      # Add specific file to staging
git add .               # Add all files to staging
git commit -m "message" # Commit staged changes
```

#### Explanation:

- `git add` – Moves files from the working directory into the staging area.
- `git add .` – Stages all files in the current directory.
- `git commit -m "message"` – Saves staged changes permanently into the repository's history.

**Tip on Messages:** Always use clear, descriptive commit messages. They serve as a logbook for your project.

Bad example:

```
git commit -m "stuff"
```

Good example:

```
git commit -m "Fixed bug in login validation"
```

## 7.3 Example Workflow

Here is a simple workflow using the commands we have learned:

### Complete Workflow Example

```
# Step 1: Initialize repository
git init

# Step 2: Check status of files
git status

# Step 3: Stage changes
git add .

# Step 4: Commit staged changes
git commit -m "Initial commit with project setup"

# Step 5: Check status again
git status
```

#### Tip

This workflow repeats many times a day. The key is to commit frequently with clear messages. Think of each commit as a “save point” in your project's history.

By practicing these essential Git commands, you will gain confidence in managing your repositories. They form the building blocks for more advanced operations like branching, merging, and collaborating on GitHub.

---

## 8 Working with VS Code and Git

**Visual Studio Code (VS Code)** is one of the most popular code editors today. It provides an integrated environment to write code, manage files, and interact with Git—all without leaving the editor.

Using VS Code with Git offers several advantages:

- A built-in terminal to run Git commands directly.
- Visual indicators for file states (Modified, Untracked, Staged).
- Extensions like GitLens that add powerful features for exploring history and commits.
- A smooth workflow for beginners and professionals alike.

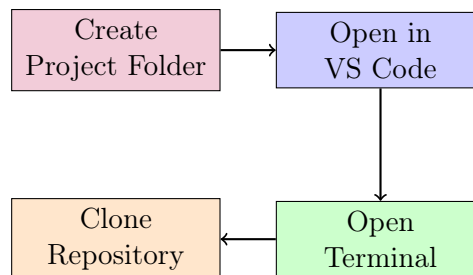


Figure 8: VS Code Git Setup Workflow

### 8.1 Project Setup

To begin working on a new project in VS Code:

1. **Create a project folder** on your computer. This folder will hold all your code and related files.
2. Open **Visual Studio Code**.
3. Click on **File** → **Open Folder** (or “Open” from the welcome screen).
4. Select your project folder. VS Code will load it into the Explorer panel on the left side.
5. Open a new terminal within VS Code:
  - Mac: **Control** + **~** (tilde key).
  - Windows: **Control** + **'** (single quote).
6. Navigate inside the terminal to confirm you are in the correct project folder:

```
pwd
```

#### VS Code Gem

Enable *Git: Auto Fetch* in VS Code Settings to keep your local branches aware of remote changes without manual pulls.

#### Tip

Always keep one VS Code window per project. This avoids confusion and ensures Git tracks files in the correct repository.



---

## 8.2 File Operations in VS Code

VS Code makes it easy to manage files alongside Git integration:

- **Creating Files:** Use the “+ New File” icon in the Explorer panel.
- **Creating Folders:** Use the “+ New Folder” icon.
- **Renaming or Deleting Files:** Right-click a file or folder in the Explorer and choose the action.

Git integration in VS Code automatically shows file states:

- **M (Modified):** A file has been changed since the last commit.
- **U (Untracked):** A new file not yet tracked by Git.
- **A (Added):** A file that has been staged for commit.

You can also view Git changes in the **Source Control panel** (left sidebar). It allows you to stage, unstage, and commit changes with buttons instead of terminal commands.

### Best Practice

Even though VS Code provides GUI buttons for staging and committing, it's recommended to learn the terminal commands. They work the same way in every environment and give you more control.

## 8.3 Cloning Repositories

Cloning is the process of copying a repository from GitHub to your local computer.

1. Go to the repository page on GitHub.
2. Click the green **Code** button.
3. Copy the **HTTPS** URL (or SSH if you have set up keys).
4. In VS Code, open the terminal and run:

```
git clone https://github.com/username/repository.git
```

5. Navigate into the cloned repository:

```
cd repository
```

6. Open the folder in VS Code if it is not already open.

### Tip

Always clone repositories into a dedicated project folder. Avoid cloning into random directories to keep your workspace organized.

Once cloned, you can:

- Edit files directly in VS Code.
- Stage and commit changes.

- 
- Push updates back to GitHub using the terminal or VS Code's Source Control panel.

Cloning is especially useful for:

- Contributing to open-source projects.
- Working on team projects hosted on GitHub.
- Keeping a local backup of your repositories.

#### Best Practice

When cloning a repository for the first time, always run:

```
git remote -v
```

to confirm that the remote URL is correct. This prevents pushing to the wrong location.

---

## 9 Core Git Concepts

Working with Git and GitHub requires understanding a few key concepts. These concepts form the foundation for how you manage projects, collaborate with others, and maintain a clean history of your work.

In this section, we cover three important ideas:

1. Local vs Remote repositories.
2. Writing good commit messages.
3. Git's tracking mechanism for changes.

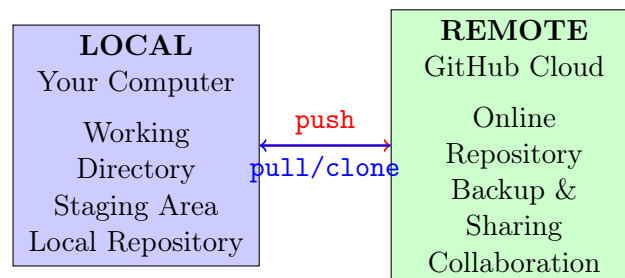


Figure 9: Local vs Remote Repositories

### 9.1 Local vs Remote

Git uses a distributed model, which means you always have a full copy of the repository on your local machine. This makes Git extremely powerful compared to centralized version control systems.

**Local repository (your computer):**

- Contains the working directory, staging area, and commit history.
- Lets you work offline and still track all changes.
- Changes can be staged and committed without an internet connection.

**Remote repository (GitHub):**

- Stored on GitHub's servers in the cloud.
- Acts as a backup for your work.
- Enables collaboration by allowing others to pull your changes and push their own.

Common commands for syncing between local and remote:

```
git push origin main    # Send local commits to remote
git pull origin main    # Fetch and merge latest changes
git clone [url]         # Copy a remote repo to local
```

#### Concept Check

You can commit everything locally without internet. Pushing/pulling is only needed to sync with a remote like GitHub.

### Best Practice

Always pull the latest changes (`git pull`) before starting new work. This prevents conflicts by ensuring your local copy is up to date.

## 9.2 Commit Messages

Every commit requires a message, and these messages are crucial for understanding your project's history. Good commit messages act as documentation and make collaboration easier.

**Guidelines for writing commit messages:**

- Use the **imperative mood**: “Add feature” not “Added feature.”
- Keep the message concise but descriptive.
- Explain the *why*, not just the *what*.
- For larger commits, use a subject line and a body.

Examples:

Bad message:

```
git commit -m "fixed stuff"
```

Good message:

```
git commit -m "Add validation for email input in signup form"
```

Better with extended description:

```
Add validation for email input in signup form
- Added regex-based validation
- Prevents users from submitting invalid emails
- Improves form error handling
```

### Tip

Think of commits as “save points” in a video game. Your commit messages are the labels that help you quickly find the right checkpoint later.

## 9.3 Git Tracking

One of Git's superpowers is its ability to track changes precisely. Unlike older systems that stored entire file versions, Git stores changes (deltas) efficiently.

**What Git tracks:**

- File additions (new files created).
- File modifications (lines changed).
- File deletions (removed files).
- Renamed or moved files (detected intelligently).

Example of Git showing tracked changes:

```
git status
```

---

```
On branch main
Changes not staged for commit:
  modified:   index.html
  deleted:    old_script.js

Untracked files:
  new_feature.py
```

Git even lets you inspect the exact line-level changes:

```
git diff
```

This command highlights what lines were added (green) or removed (red).

### Best Practice

Commit often and in small chunks. This way, Git's tracking history is detailed, and reverting mistakes becomes much easier.

By mastering these core concepts—local vs remote, clear commit messages, and Git tracking—you build strong habits that make you an efficient and reliable developer.

## 10 Quick Reference

This section serves as a handy cheat sheet for the most essential Git and terminal commands. Keep it nearby when working on projects for quick recall.

### Essential Commands

```
# --- Setup (one-time configuration) ---
git config --global user.name "Your Name"
git config --global user.email "your@email.com"
git config --global init.defaultBranch main

# --- Daily Workflow ---
git status          # Check current state of files
git add .           # Stage all changes
git commit -m "message" # Commit staged changes
git push origin main # Push changes to remote
git pull origin main # Pull latest changes from remote

# --- Repository Operations ---
git init            # Initialize a new repository
git clone [url]     # Copy a remote repo locally

# --- File Navigation (terminal basics) ---
pwd                # Print working directory
ls                 # List files and folders
ls -a              # List all files, including hidden
cd folder          # Move into folder
cd ..              # Move one level up
mkdir folder_name  # Create new folder
clear              # Clear terminal screen

# --- Inspecting Changes ---
git log            # View commit history
git diff           # See line-by-line changes
git show [commit_id] # Show details of a specific commit

# --- Help ---
git help           # Open general help
git help [command] # Help for a specific command
```

### Speed Tip

`git status -sb` gives a short, clean summary. Also try `git <command> -h` for quick inline help.

### Tip

If you ever forget a command, run: `git help` or `git help <command>`  
For example: `git help commit`

With these commands, you can confidently navigate the basics of Git and GitHub. As you grow more experienced, you will explore advanced commands like branching, merging, and rebasing—but these essentials will always remain at the heart of your workflow.