



Karan Gupta

# Source Code Management



# Source Code Management

- **Definition:**

- Source Code Management (SCM) is the practice of tracking and managing changes to software code.

- **Purpose:**

- Helps in maintaining a history of changes.
  - Facilitates collaboration among developers.
  - Ensures consistency and quality of the codebase.

# Key Practices and Tools in SCM

- **Key Practices:**

- Version Control: Using tools like Git to track changes.
- Branching and Merging: Creating branches for features and merging them back into the main codebase.
- Code Review: Peer review of code changes to maintain quality.

- **Popular Tools:**

- Git
- Subversion (SVN)
- Mercurial



# Benefits of Effective Source Code Management

- **Improved Collaboration:**
  - Multiple developers can work on the same project simultaneously without conflicts.
- **Enhanced Code Quality:**
  - Regular code reviews and testing ensure high-quality code.
- **Better Project Management:**
  - Track progress, manage releases, and maintain historical versions.
- **Disaster Recovery:**
  - Ability to revert to previous versions in case of issues.



# Relation in VCS - SCM





Karan Gupta

# Relation between VCS – SCM

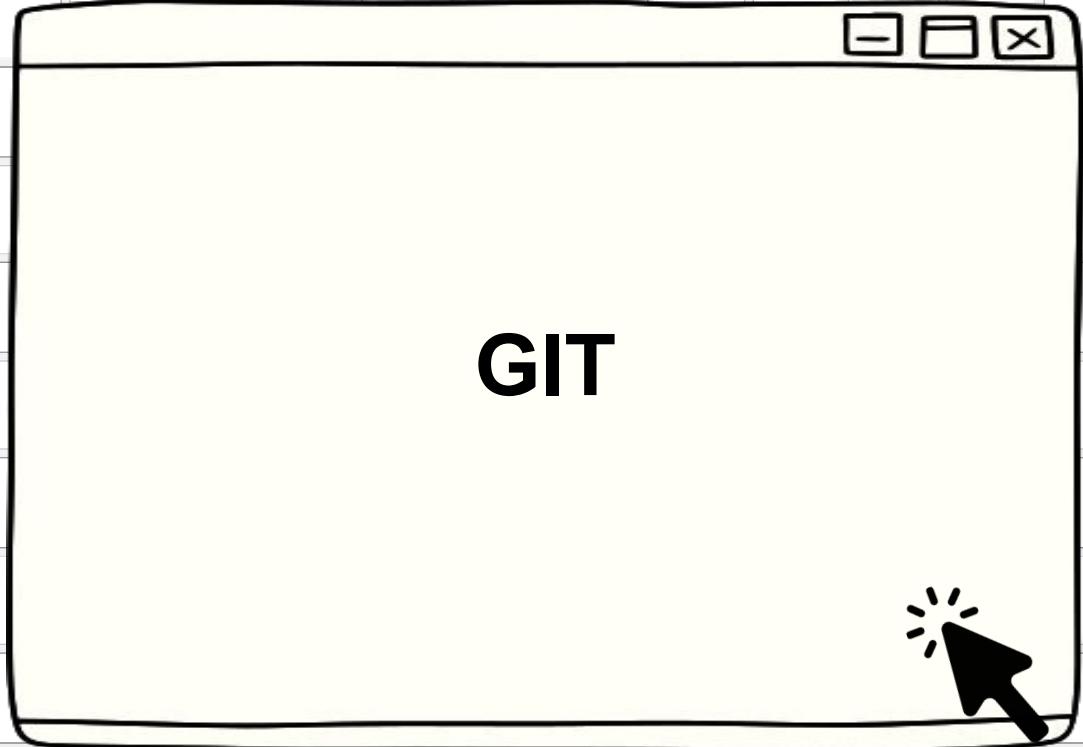
- Version Control System (VCS) is a **subset** of Source Code Management (SCM).
- **VCS tracks code changes**, while **SCM manages collaboration, workflows, and integrations**.

## Example in Git Context:

- **Git as a VCS** → Tracks commits, branches, and merges.
- **GitHub/GitLab as SCM** → Provides collaboration, pull requests, CI/CD, and permissions.

## Key Takeaways:

- VCS is essential for tracking changes but SCM extends it for team collaboration and automation.
- SCM = VCS + Collaboration + Automation.



# Introduction to Git

- **Definition:**

- Git is a distributed version control system (VCS) designed to handle everything from small to very large projects with speed and efficiency.

- **Purpose:**

- Tracks changes to source code.
- Facilitates collaboration among developers.
- Manages project versions and history.

# Key Features of Git

- **Distributed System:**
  - Every developer has a full copy of the repository.
- **Performance:**
  - Optimized for speed and efficiency.
- **Branching and Merging:**
  - Lightweight branching and merging to support parallel development.
- **Data Integrity:**
  - Ensures data is not corrupted during transit or storage.
- **Support for Non-linear Development:**
  - Ability to work on multiple tasks simultaneously and merge changes as needed.





Karan Gupta



# Git in Modern Software Development

- **Widespread Adoption:**

- Used by major tech companies like Google, Microsoft, and Facebook.
- Integral to many open-source projects.

- **Collaboration:**

- Facilitates teamwork and parallel development.
- Supports distributed teams working across different geographies.

- **Efficiency:**

- Speeds up development cycles through efficient version control and branching.

# Benefits of Using Git in Industry

- **Version Control:**
  - Keeps track of every change made to the codebase.
  - Allows reverting to previous states easily.
- **Branching and Merging:**
  - Supports multiple development workflows (e.g., feature branches, hotfixes).
  - Facilitates code integration from different branches.
- **Integration with CI/CD:**
  - Seamlessly integrates with Continuous Integration/Continuous Deployment pipelines.
  - Automates testing and deployment processes.



Karan Gupta

# Real-World Examples and Case Studies

- **Open Source Projects:**

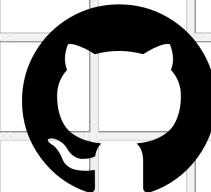
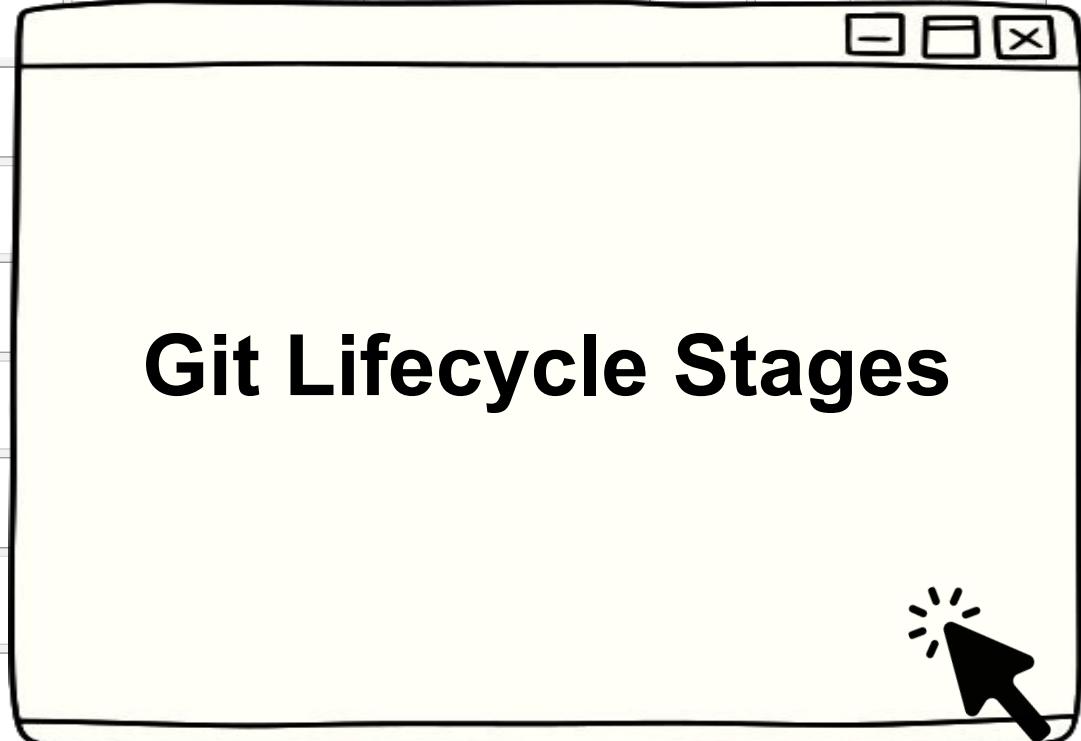
- Linux Kernel: Coordinated by thousands of contributors worldwide.
- Kubernetes: Managed through Git for consistent and reliable updates.

- **Enterprise Use:**

- Microsoft: Uses Git for the development of its products and services.
- Facebook: Migrated to Git for its scale and efficiency.

- **Case Study:**

- Example of a company improving its development workflow by adopting Git.
- Metrics showing reduced deployment time and increased productivity.



# Git Stages:

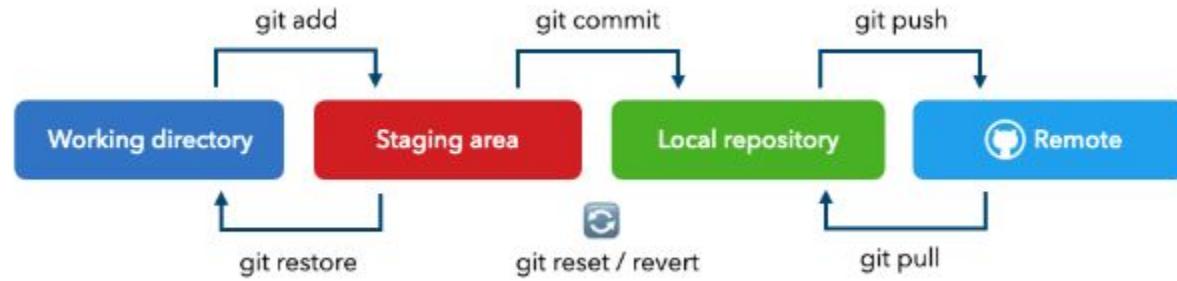
Git has **three stages** that track changes in your project:

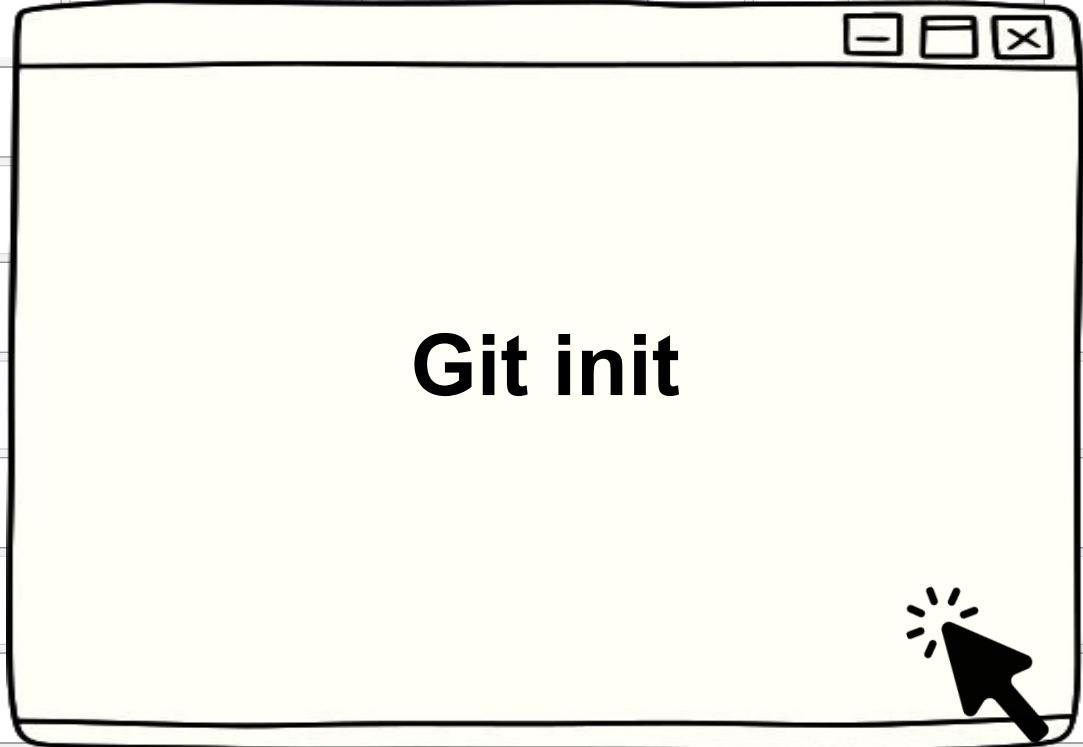
1. **Working Directory** – Where you edit files.
2. **Staging Area** – Where you prepare files for commit.
3. **Repository (Local/Remote)** – Where committed changes are stored.

## Git Workflow Overview

- 1 Modify a file in the **Working Directory**.
- 2 Add it to the **Staging Area** using **git add**.
- 3 Commit it to the **Repository** using **git commit**.







## **Overview:**

- Initializing a Git repository is the first step to start tracking changes in a project.

## **Command:**

- `git init`: Creates a new Git repository.

## **Steps to Initialize:**

1. Open the terminal.
2. Navigate to your project directory.
3. Run the command `git init`.

## **Post Initialization:**

- A `.git` directory is created in the project folder.
- This directory contains all the metadata and version history for the project.

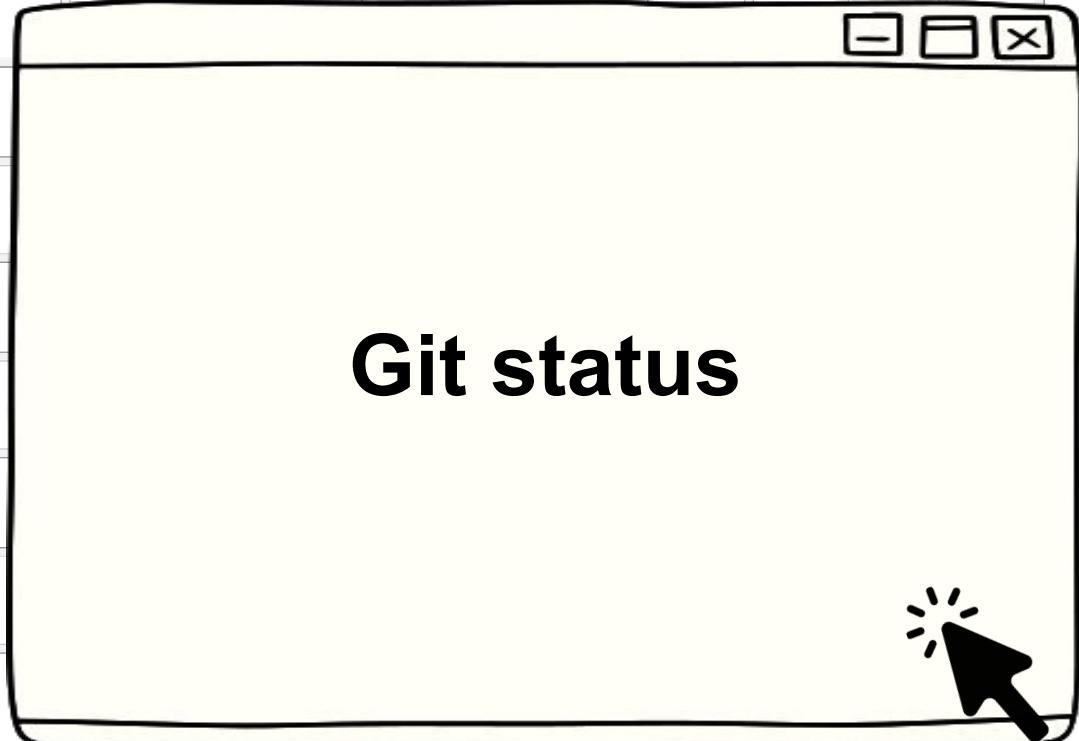
shell

 Copy code

```
$ cd my-project
```

```
$ git init
```

```
Initialized empty Git repository in /path/to/my-project/.git/
```



# Understanding `git status` Command

## Overview:

- `git status` provides information about the current state of the working directory and staging area.

## Purpose:

- Shows which changes have been staged, which haven't, and which files aren't being tracked by Git.

## Command:

- Run `git status` in the terminal to view the status.

## Output Explanation:

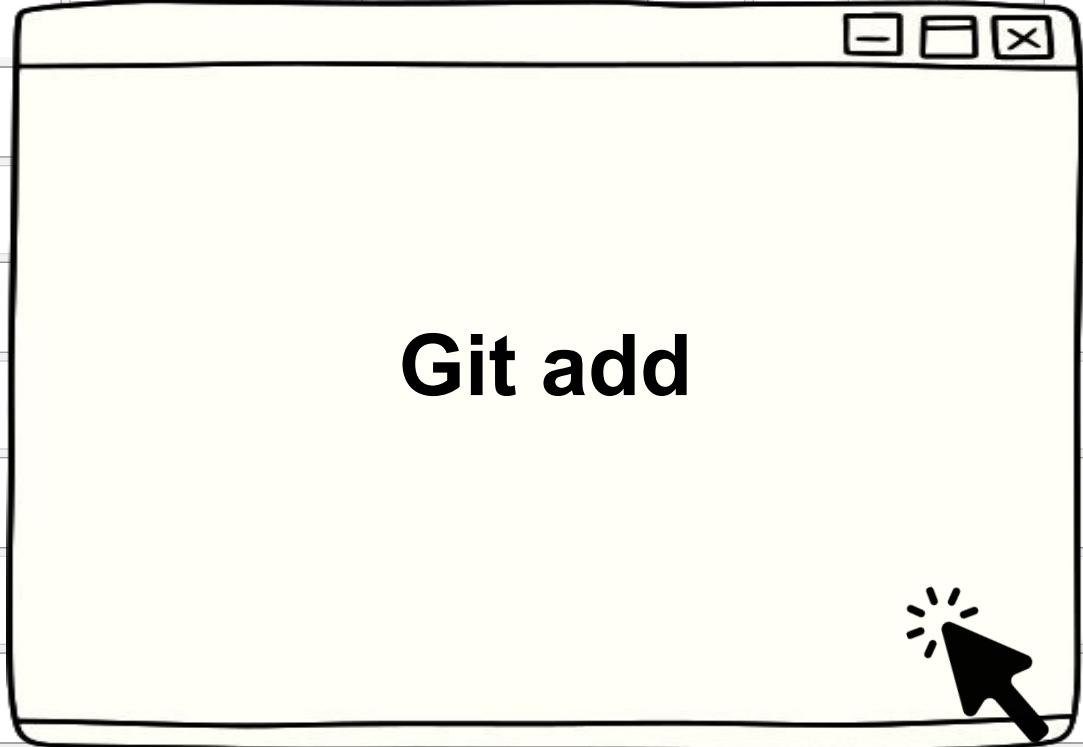
- **Untracked files:** Files in the working directory that are not yet being tracked by Git.
- **Changes not staged for commit:** Modified files that are not yet staged.
- **Changes to be committed:** Files that have been staged and are ready to be committed.

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newfile.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    modified:   file1.txt

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   file2.txt
```



# Understanding `git add` Command

- `git add` is used to add changes in the working directory to the staging area.

## Purpose:

- Prepares changes to be included in the next commit.
- Tracks new files and stages modifications or deletions of existing files.

## Command Usage:

- `git add <file>`: Stages a specific file.
- `git add .` : Stages all changes in the current directory and subdirectories.

```
$ git add README.md  
$ git add .  
$ git add -A
```



# Understanding `git commit` Command

## 1. Overview:

- `git commit` saves the changes staged in the staging area to the local repository.
- Each commit represents a snapshot of the project at a specific point in time.

## 2. Purpose:

- Records a complete history of changes made to the project.
- Allows for tracking progress, reverting to previous states, and collaboration.

## 3. Command Usage:

- `git commit -m "commit message"`: Creates a commit with a descriptive message.

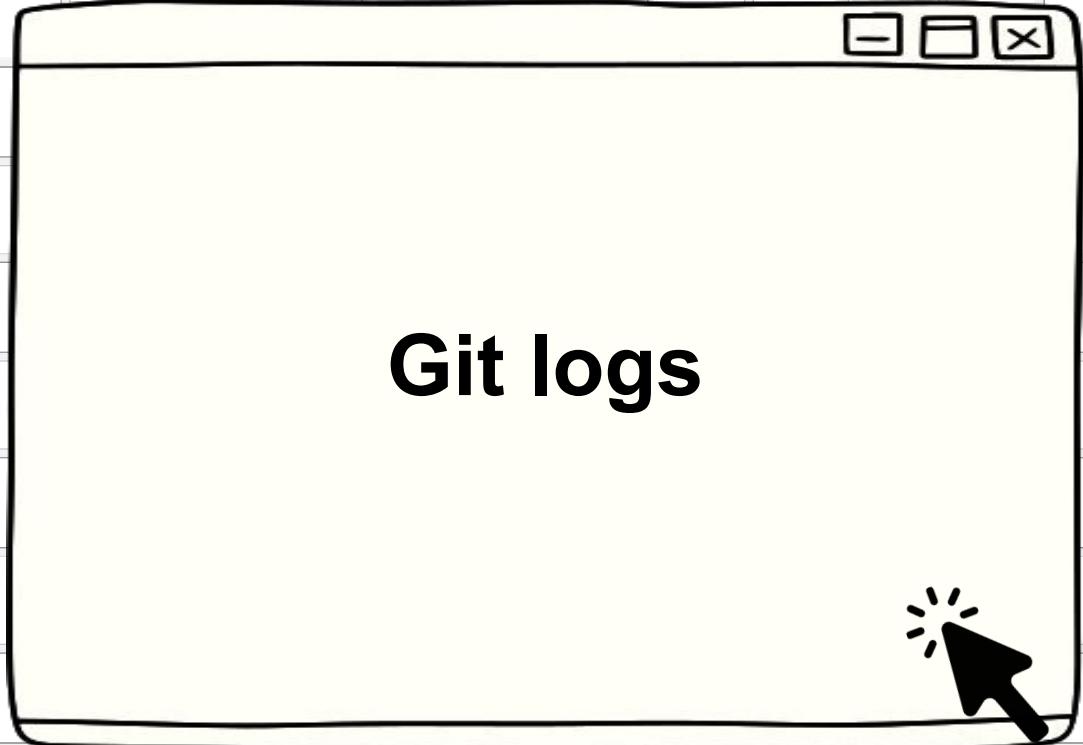
## 4. Commit Message Best Practices:

- Keep messages concise and descriptive.

shell

 Copy code

```
$ git commit -m "Add README file"
[master (root-commit) 83d16e4] Add README file
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```



# Understanding `git logs` Command

- `git log` displays the commit history for the repository.
- Provides details about each commit, including the author, date, and commit message.

## Purpose:

- Helps in tracking changes and understanding the project's history.
- Useful for debugging and auditing changes.

## Command Usage:

- Basic command: `git log`
- Common options:
  - `git log --oneline`: Shows a simplified, one-line-per-commit view.
  - `git log -p`: Shows the changes introduced in each commit.
  - `git log --stat`: Shows statistics for files modified in each commit.

```
$ git log
```

```
commit 83d16e48d9a58c3c0280a4c8b4c4d5b7a5e1d14b
```

```
Author: John Doe <johndoe@example.com>
```

```
Date: Mon Jul 19 15:35:42 2024 +0100
```

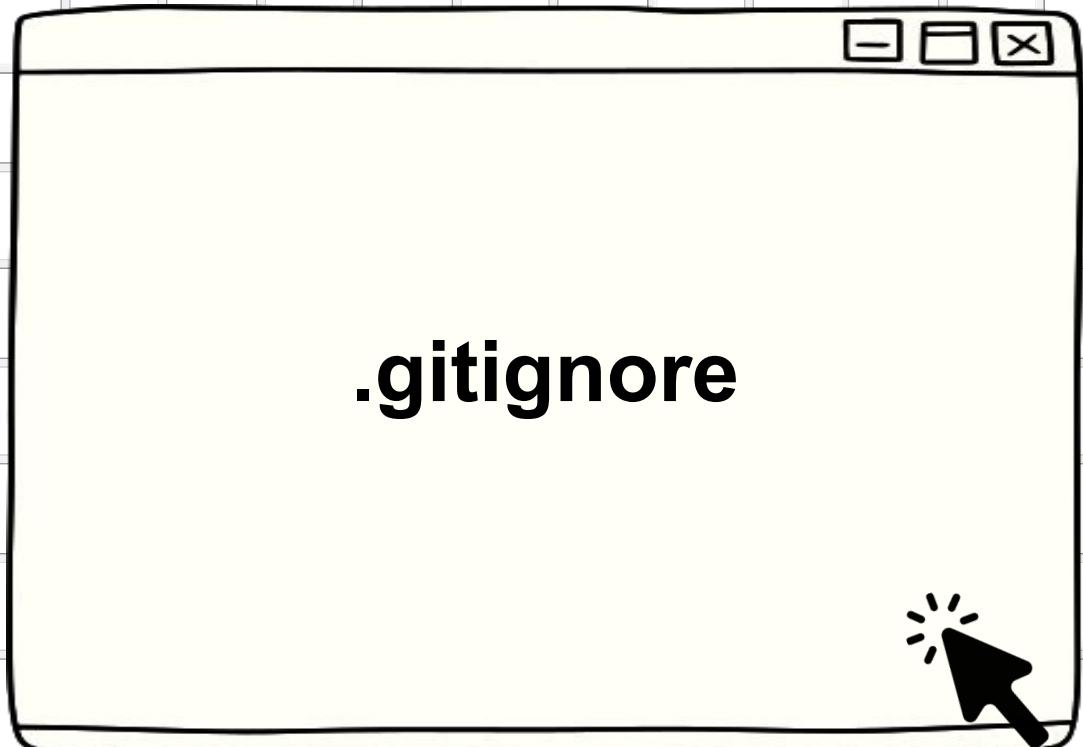
```
    Add README file
```

```
commit f6d8a2d4c3d5f6e8b7c8d9e0f0a1b2c3d4e5f6a7
```

```
Author: Jane Smith <janesmith@example.com>
```

```
Date: Sun Jul 18 10:20:30 2024 +0100
```

```
Initial commit
```



# Understanding .gitignore

- The `.gitignore` file tells Git which files (or patterns) it should ignore and not track.

## Purpose:

- Prevents certain files from being included in version control.
- Commonly used for files that are generated during the build process, sensitive information, or files that are specific to a local environment.

## Creating a `.gitignore` File:

- Create a file named `.gitignore` in the root directory of your project.
- List the files and directories you want to ignore.

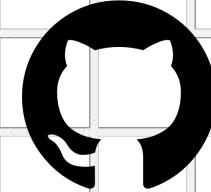
## Syntax:

- Each line in the `.gitignore` file specifies a pattern.
- Examples:
  - `*.log`: Ignores all `.log` files.
  - `build/`: Ignores the `build` directory.
  - `config/settings.py`: Ignores a specific file.

## Common Use Cases:

- Ignoring operating system files (e.g., `Thumbs.db`, `.DS_Store`).
- Ignoring dependency directories (e.g., `node_modules`, `venv`).
- Ignoring configuration files (e.g., `*.env`, `config.yml`).

```
# Ignore all log files  
*.log  
  
# Ignore the build directory  
build/  
  
# Ignore a specific file  
config/settings.py
```



# Git clone

- `git clone` is used to create a local copy of a remote repository.
- Copies the entire repository, including all files, history, and branches.

## Purpose:

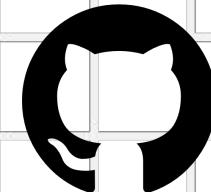
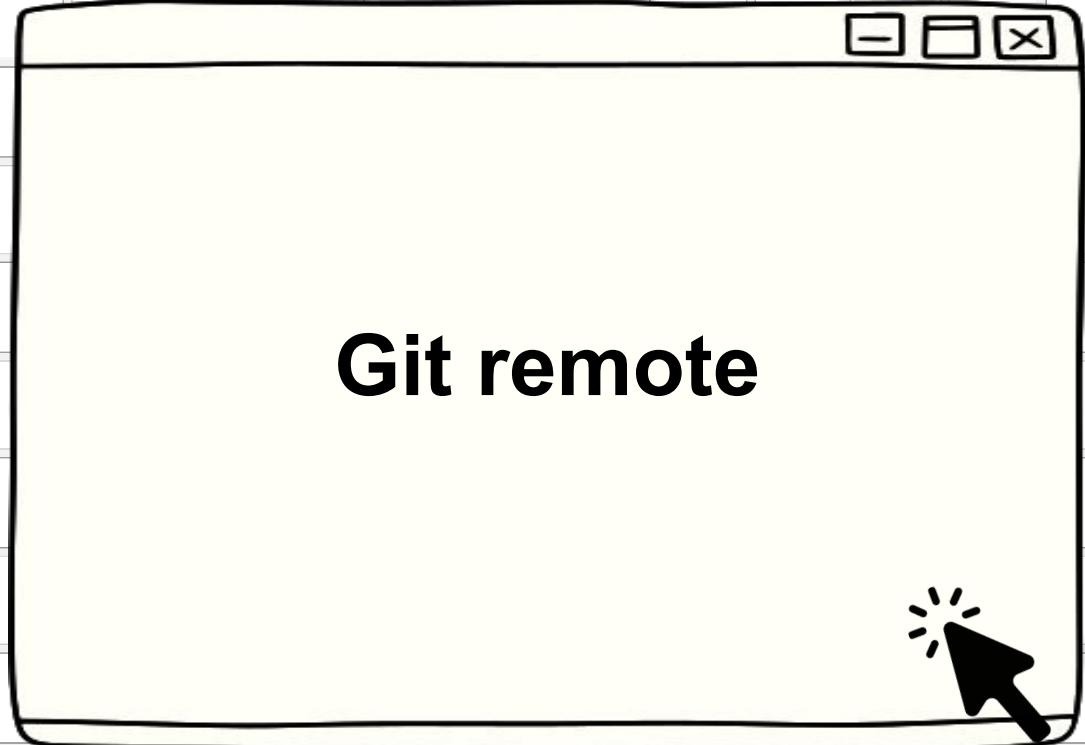
- Provides a local working copy of a repository to start working with.
- Essential for collaborating on projects hosted on platforms like GitHub, GitLab, or Bitbucket.

## Command Usage:

- Basic command: `git clone <repository-url>`



```
karangupta@KaranGupta new % git clone https://github.com/CourseDIY/first-resource.git
Cloning into 'first-resource'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (1/1), done.
karangupta@KaranGupta new % cd first-resource
karangupta@KaranGupta first-resource % ls
FirstFile      README.md      service.go
karangupta@KaranGupta first-resource %
```



# Git remote

- `git remote add` is used to add a new remote repository to your local Git repository.
- A remote repository is a version of your project hosted on a server (e.g., GitHub, GitLab, Bitbucket).

## Purpose:

- Connects your local repository to a remote server for collaboration and version control.
- Enables you to push and pull changes to and from the remote repository.

## Command Usage:

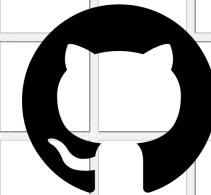
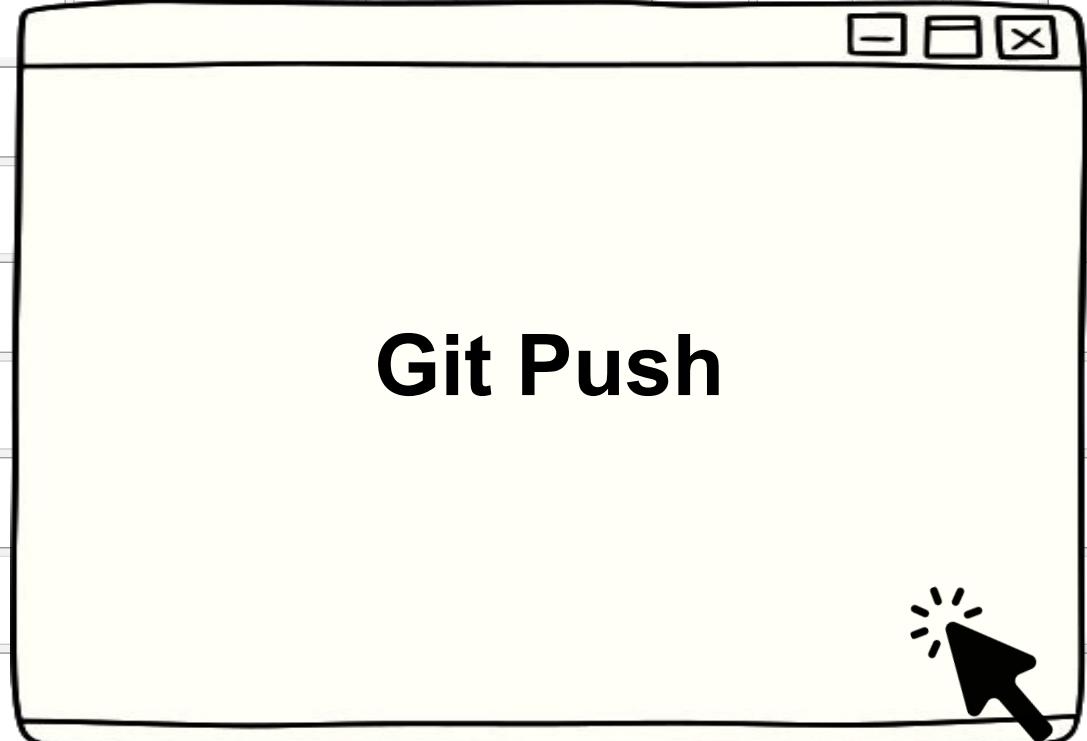
- Basic command: `git remote add <name> <repository-url>`
- `<name>`: A short name for the remote repository (e.g., `origin`).
- `<repository-url>`: The URL of the remote repository (e.g., <https://github.com/user/repository.git>).

## Viewing and Managing Remotes:

- `git remote -v`: Lists all configured remotes and their URLs.
- `git remote remove <name>`: Removes a remote.
- `git remote rename <old-name> <new-name>`: Renames a remote.



```
karangupta@KaranGupta git-project %
karangupta@KaranGupta git-project % git remote add origin https://github.com/CourseDIY/first-resource.git
karangupta@KaranGupta git-project %
karangupta@KaranGupta git-project % git remote -v
origin https://github.com/CourseDIY/first-resource.git (fetch)
origin https://github.com/CourseDIY/first-resource.git (push)
karangupta@KaranGupta git-project % █
```



# Git Push

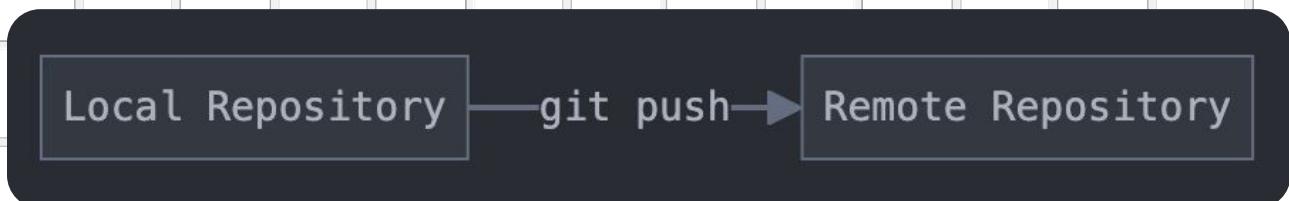
The `git push` command is used to upload local repository content to a remote repository.

## Purpose:

- Shares changes made locally with teammates or collaborators.
- Keeps the remote repository up to date.

## How It Works:

1. Make changes locally.
2. Commit changes (`git commit`).
3. Push changes to the remote repository (`git push`).



- **Why Use It?**

- a. Shares your work with others in real-time.
- b. Ensures the remote repository is synchronized with your local changes.

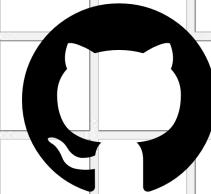
- **When to Use It?**

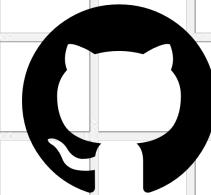
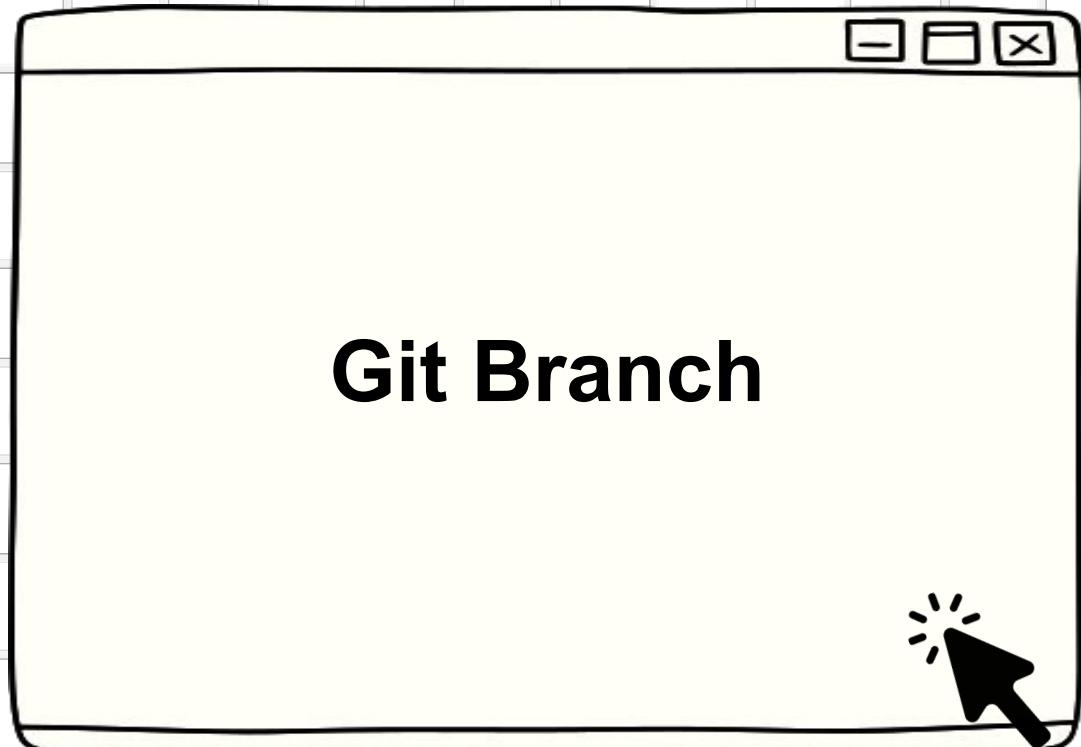
- a. After making commits locally.
- b. When you're ready to share updates or collaborate with others.

- **Example:**

You've added new features to a project on your laptop. Use `git push` to upload those updates to a shared GitHub repository so teammates can access them.





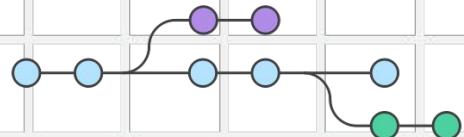


# Git Branch

- A branch in Git is a lightweight, movable pointer to a commit. It allows you to work on different versions of a project simultaneously.

## Key Features:

- Facilitates parallel development.
  - Keeps the main branch stable while testing features.
  - Allows team collaboration by isolating changes.



## Types of Branches:

- **Main/Default Branch:** Typically `main` or `master`.
  - **Feature Branch:** For new features.
  - **Bugfix Branch:** For fixing issues.
  - **Release Branch:** For preparing releases.

## **Command Usage:**

- **List Branches:**
  - `git branch`: Lists all branches in the local repository.
  - `git branch -a`: Lists all branches, including remote branches.
- **Create a Branch:**
  - `git branch <branch-name>`: Creates a new branch named `<branch-name>`.
- **Delete a Branch:**
  - `git branch -d <branch-name>`: Deletes the specified branch (only if it has been fully merged).
  - `git branch -D <branch-name>`: Forcefully deletes the branch (use with caution).
- **Rename a Branch:**
  - `git branch -m <old-name> <new-name>`: Renames a branch from `<old-name>` to `<new-name>`.

```
karangupta@KaranGupta first-resource % git branch
* main
karangupta@KaranGupta first-resource % git branch -v
* main a85806f Add files via upload
karangupta@KaranGupta first-resource % git branch test
karangupta@KaranGupta first-resource % git branch -v
* main a85806f Add files via upload
  test a85806f Add files via upload
karangupta@KaranGupta first-resource % git checkout test
Switched to branch 'test'
karangupta@KaranGupta first-resource % git branch -v
  main a85806f Add files via upload
* test a85806f Add files via upload
karangupta@KaranGupta first-resource % git branch -d test
error: cannot delete branch 'test' used by worktree at '/Users/karangupta/new/first-resource'
karangupta@KaranGupta first-resource % git checkout main
Switched to branch 'main'
karangupta@KaranGupta first-resource % git branch -d test
Deleted branch test (was a85806f).
```



# git checkout feature-xyz

## Content:

## Overview:

- `git checkout` is used to switch between branches or restore working directory files to a specific state.
- Essential for navigating different lines of development or undoing changes.

## Purpose:

- Switch to a different branch.
- Restore files to a specific commit or branch.
- Create a new branch and switch to it in a single command.

## Command Usage:

- **Switch Branches:**
  - `git checkout <branch-name>`: Switches to the specified branch.
  -

## Create and Switch to a New Branch:

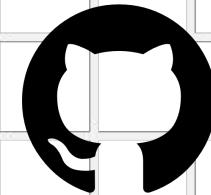
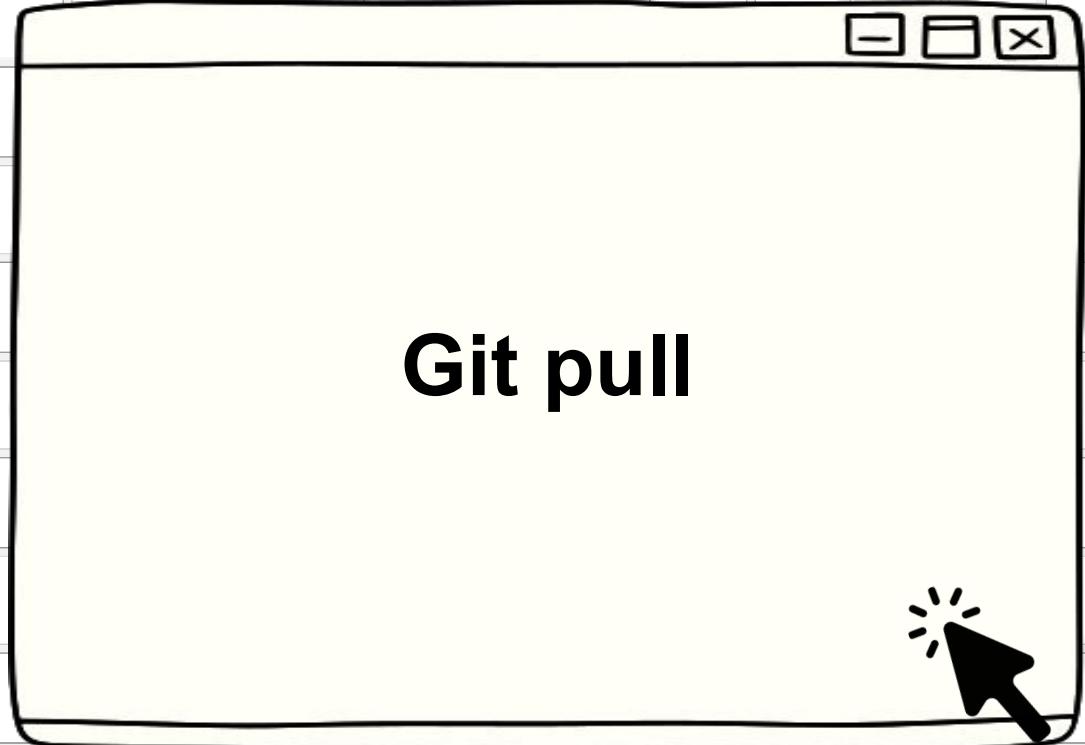
- `git checkout -b <new-branch-name>`: Creates a new branch and switches to it immediately.
- 

```
git checkout -b new-feature
```

## Restore Files to a Specific Commit:

- `git checkout <commit-hash> -- <file-path>`: Restores a file from a specific commit.
- 

```
git checkout abc123 -- README.md
```



# Git Pull



- `git pull` is used to update your local repository with changes from a remote repository.
- It performs a `git fetch` followed by a `updating of local repository` to bring the local branch up to date with the remote branch.

## Purpose:

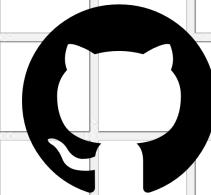
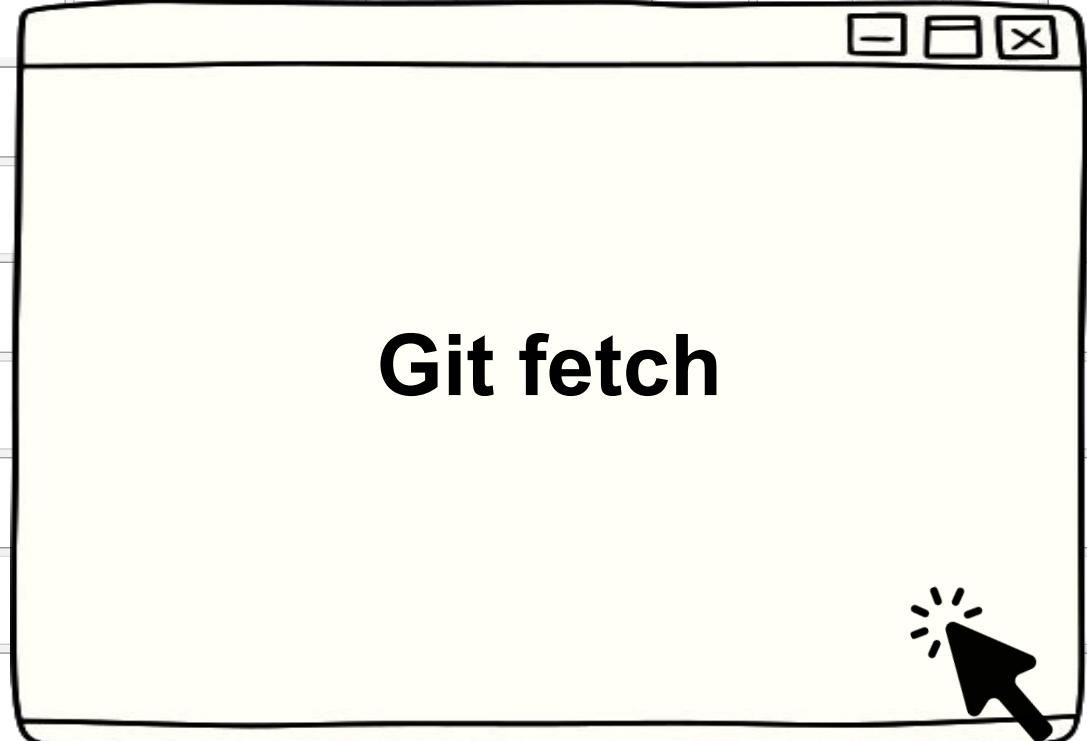
- Integrates remote changes into your local repository.
- Ensures that your local branch reflects the latest changes from the remote repository.

## Command Usage:

- Basic command: `git pull <remote> <branch>`
- `<remote>`: The name of the remote repository (e.g., `origin`).
- `<branch>`: The name of the branch to pull changes from (e.g., `main`).



```
karangupta@KaranGupta ABC % git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 978 bytes | 489.00 KiB/s, done.
From https://github.com/jadugarmjadugar/ABC
  0f1f2e1..ef96df8  main      -> origin/main
karangupta@KaranGupta ABC % git diff main origin/main
diff --git a/pulltest b/pulltest
new file mode 100644
index 0000000..976d42d
--- /dev/null
+++ b/pulltest
@@ -0,0 +1 @@
+pulling file
karangupta@KaranGupta ABC % git pull origin
Updating 0f1f2e1..ef96df8
Fast-forward
 pulltest | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 pulltest
karangupta@KaranGupta ABC % git diff main origin/main
karangupta@KaranGupta ABC %
```



# Git Fetch

- `git fetch` is used to retrieve updates from a remote repository without merging those changes into your local branch.
- It updates your remote-tracking branches, allowing you to review changes before integrating them.

## Purpose:

- Fetches changes from the remote repository and updates the remote-tracking branches in your local repository.
- Keeps your local repository aware of remote changes without modifying your current working branch.

## Command Usage:

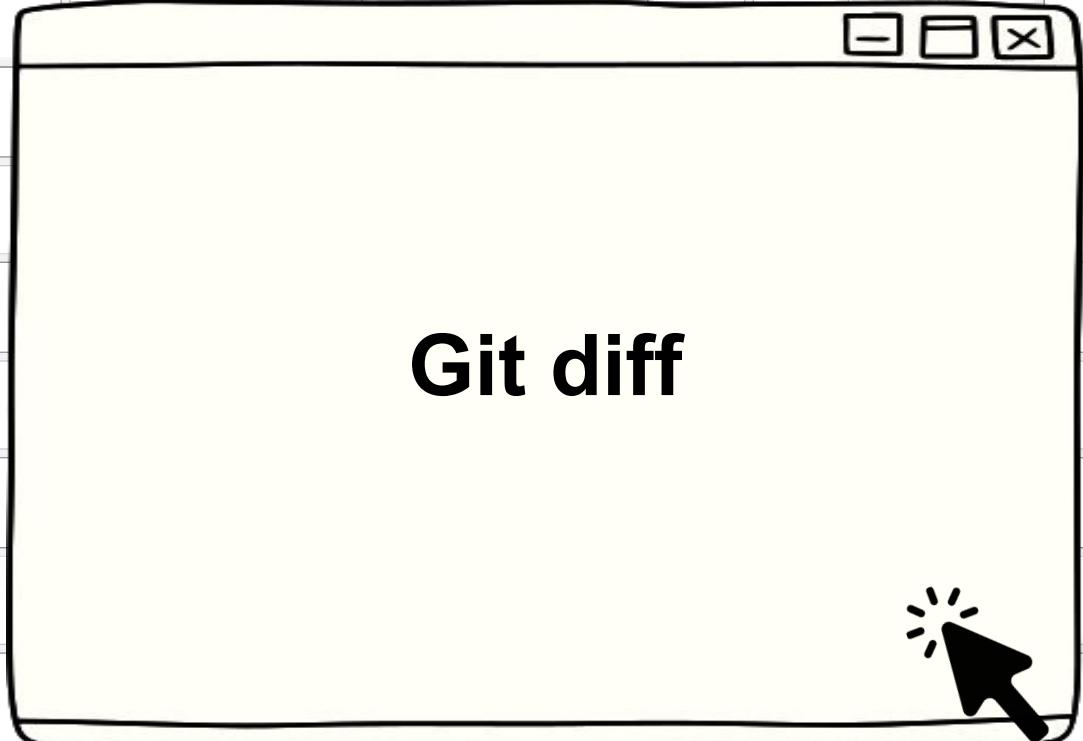
- Basic command: `git fetch <remote>`
- `<remote>`: The name of the remote repository (e.g., `origin`).



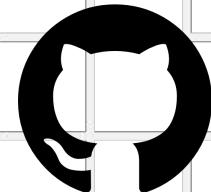
## Examples:

- `git fetch`: Fetches updates from the default remote (`origin`).
- `git fetch origin`: Fetches updates from the `origin` remote.

```
karangupta@KaranGupta ABC % git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 923 bytes | 461.00 KiB/s, done.
From https://github.com/jadugarmjadugar/ABC
      be13489..bbccfa8  main    -> origin/main
karangupta@KaranGupta ABC % git diff main origin/main
diff --git a/new b/new
new file mode 100644
index 0000000..7898192
--- /dev/null
+++ b/new
@@ -0,0 +1 @@
+a
```



git

A red diamond-shaped icon containing a white Git logo symbol (two white lines forming a branching structure) is positioned next to the word "git". The word "git" is written in a large, bold, black sans-serif font.

Karan Gupta



# Git diff



- `git diff` is used to show changes between commits, branches, files, and more.
- Provides a line-by-line comparison of differences in file content.

## Purpose:

- Review changes before committing or merging.
- Identify differences between various states of the repository.

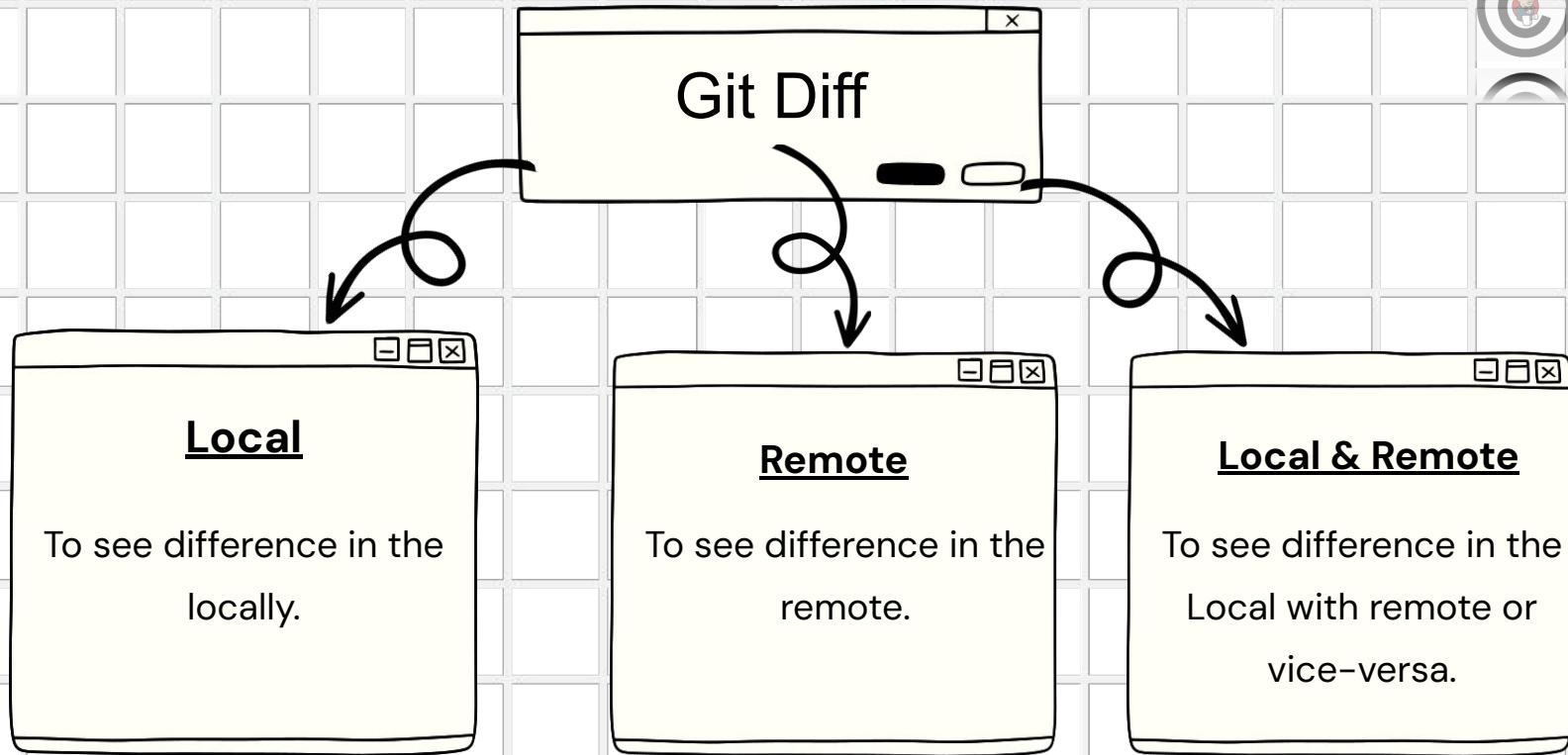
## Command Usage:

- **Compare Staging Area with Last Commit:**
  - `git diff --staged`: Shows changes that are staged for commit compared to the last commit.
- **Compare Two Commits:**
  - `git diff <commit1> <commit2>`: Shows differences between two specific commits.
- `git diff <branch1> <branch2>`: Shows differences between two branches.





Karan Gupta

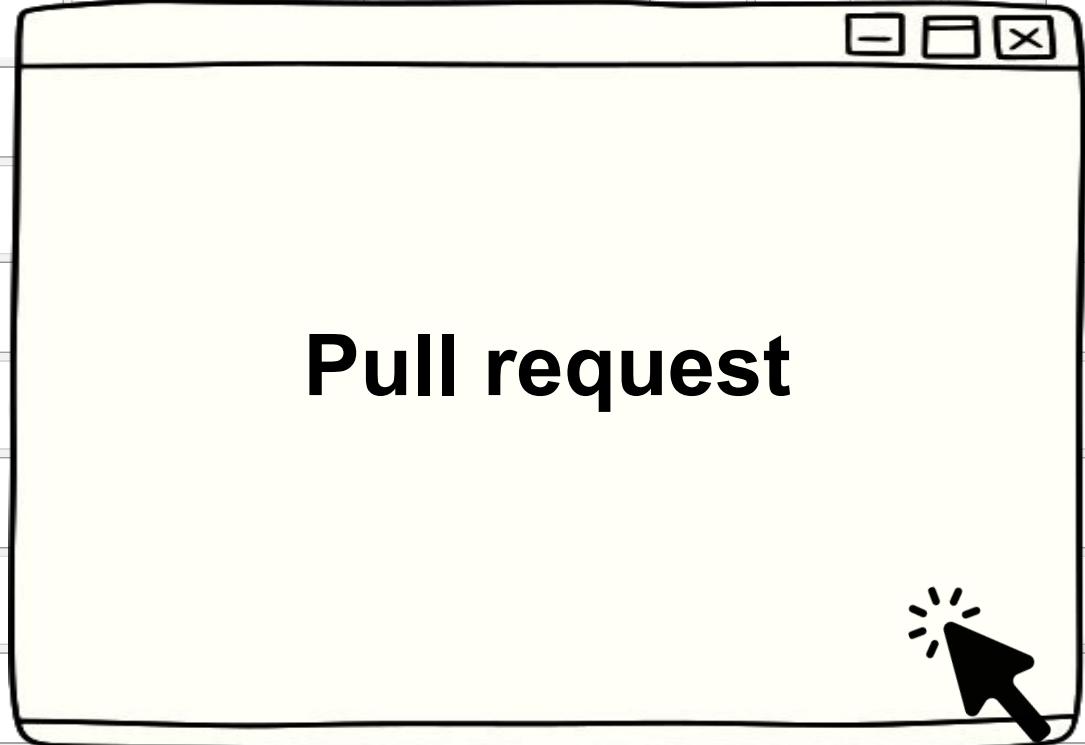




Karan Gupta



```
karangupta@KaranGupta first-resource % git branch
  main
* test
karangupta@KaranGupta first-resource % git diff main test
diff --git a/testfile b/testfile
new file mode 100644
index 0000000..7c4c185
--- /dev/null
+++ b/testfile
@@ -0,0 +1 @@
+This is my test repo
```



# Pull Request:

- A pull request (PR) is a request to merge changes from one branch into another, typically from a feature branch into the main branch.
- Used to review code before integration into the main codebase.

## Purpose:

- Allows **code review** before merging.
- Enables **team collaboration** with comments and feedback.
- Helps maintain **code quality and consistency**.
- Can be **merged after approval** by reviewers.



## Workflow:

1. Checkout to the feature branch and commit changes

```
git add .
```

```
git commit -m "Added new feature xyz"
```

2. Push the branch to remote

```
git push origin feature-xyz
```

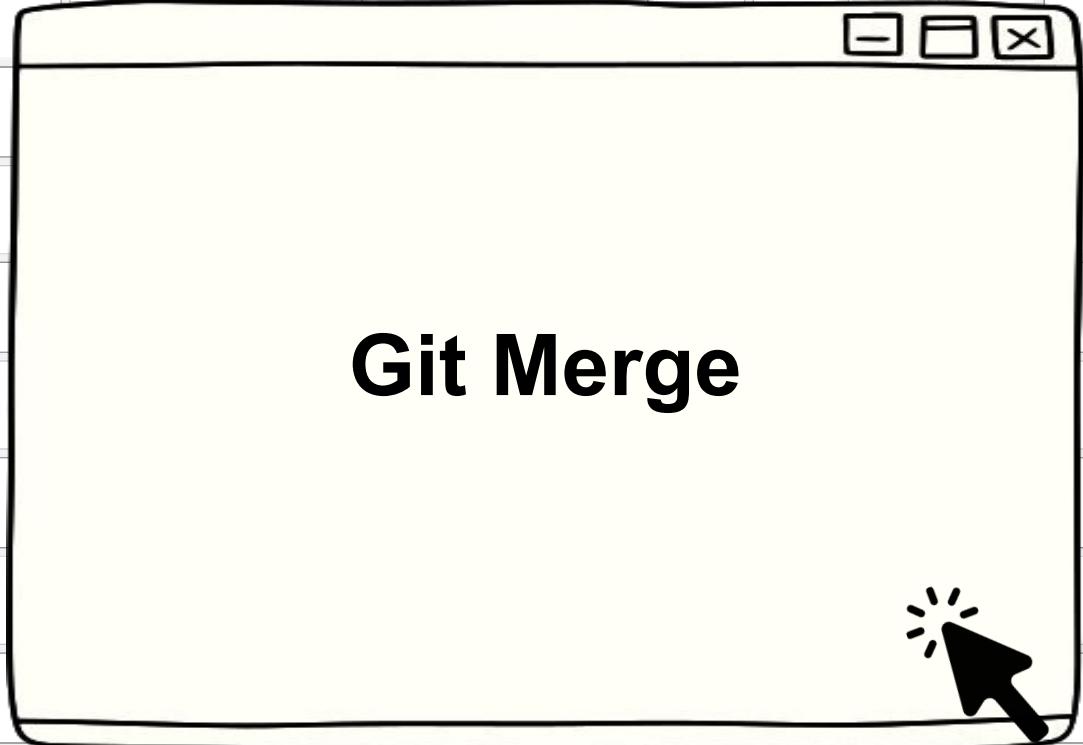
3. Go to GitHub and navigate to **Pull Requests** tab.

4. Click "**New Pull Request**", select **feature-xyz** as the source branch and **main** as the target branch.

5. Add a title and description, then click "**Create Pull Request**".

6. Wait for review, address comments, and once approved, click "**Merge**".





# Git Merge



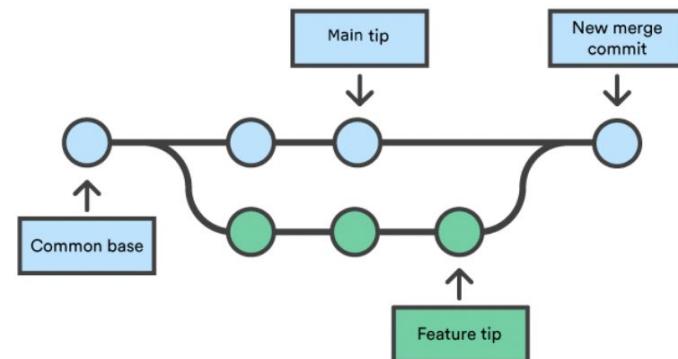
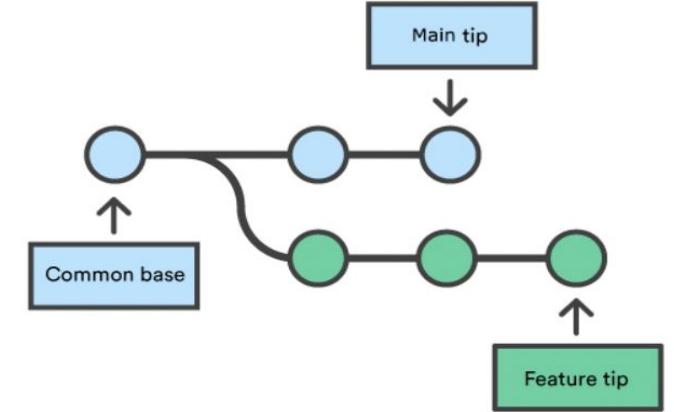
## What is Git Merge?

- o `git merge` combines changes from one branch into another.
- o Used to integrate feature development into the main branch.
- o Preserves commit history, keeping track of all changes.



## Types of Merging:

1. **Fast-forward merge** (No diverging history, moves branch pointer forward).
2. **Three-way merge** (Branches have diverged, Git creates a new merge commit).





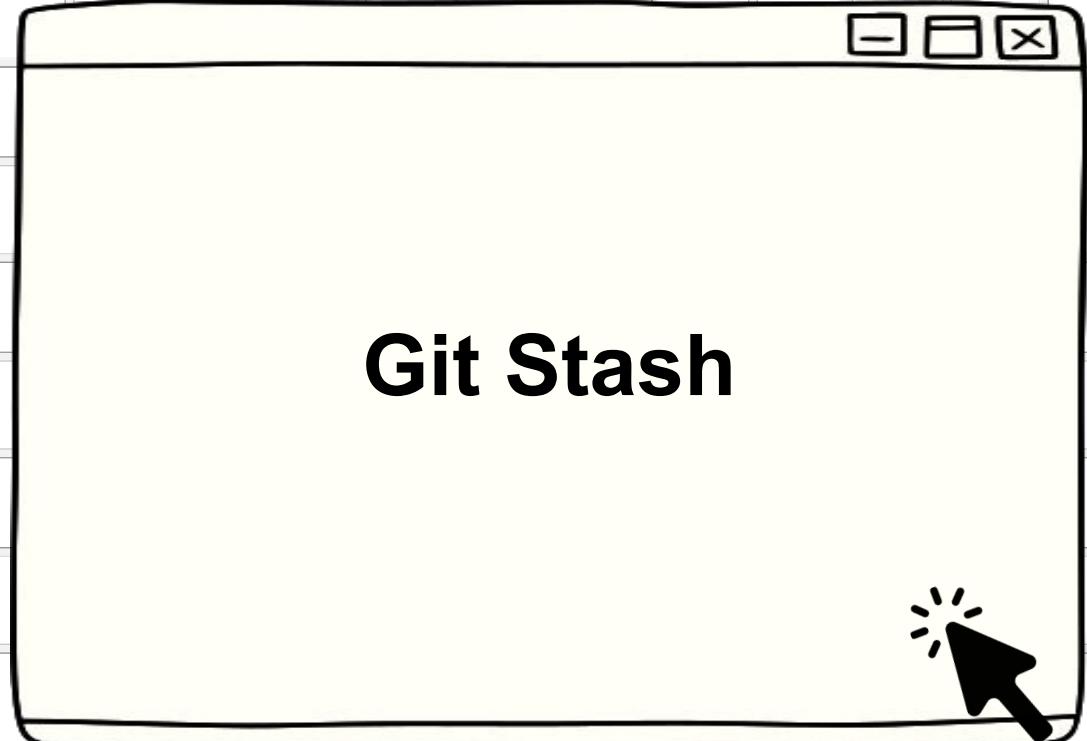
Karan Gupta

## Basic Merge Workflow:

- ① Switch to the target branch (`main` or `develop`).
- ② Use `git merge <branch>` to integrate changes.
- ③ Resolve merge conflicts if needed.
- ④ Push merged changes to the remote repository.

## Best Practices:

- **Merge Frequently:** Minimize conflicts by integrating changes regularly.
- **Review Changes:** Use `git diff` to review differences before merging.
- **Resolve Conflicts Carefully:** Ensure all conflicts are resolved correctly.
- **Test After Merging:** Verify functionality after the merge.



# Git Stash



## What is Git Stash?

- `git stash` temporarily **saves uncommitted changes** without committing them.
- Useful when you need to switch branches without losing work.
- Keeps the working directory clean without committing unfinished changes.

## Why use Git Stash:

1. Need to switch branches but don't want to commit yet.
2. Working on a feature and must apply urgent fixes elsewhere.
3. Keep local changes safe while updating the branch.

# Git Stash Practical Usage:

## 1. Save Changes Temporarily:

```
git stash
```

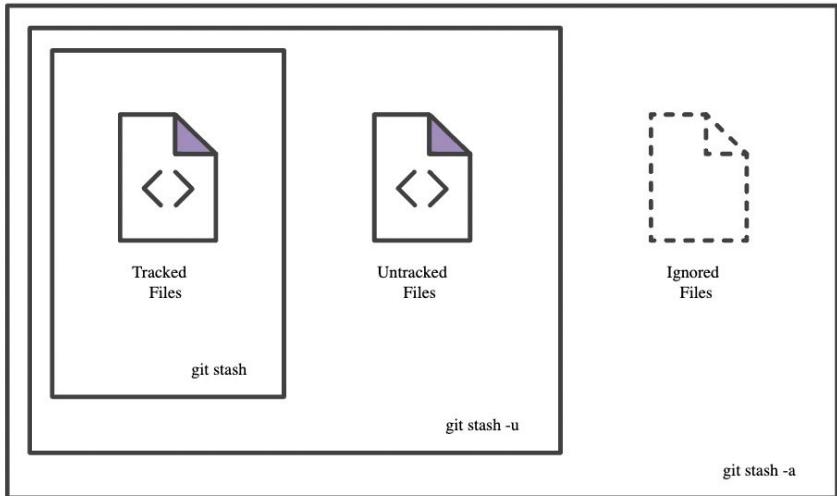
## 2. Stash with a Custom Message:

```
git stash push -m "WIP: Fixing bug in API"
```

## 3. View Stashed Change:

```
git stash list
```

git stash options



**4. Apply the Latest Stash Without Deleting It** – Reapplies changes but retains the stash entry.

```
git stash apply
```

**5. Apply and Remove the Stash** – Restores the most recent stash and deletes it from the list.

```
git stash pop
```

**6. Apply a Specific Stash** – Restores a specific stash from the list.

```
git stash apply stash@{2}
```



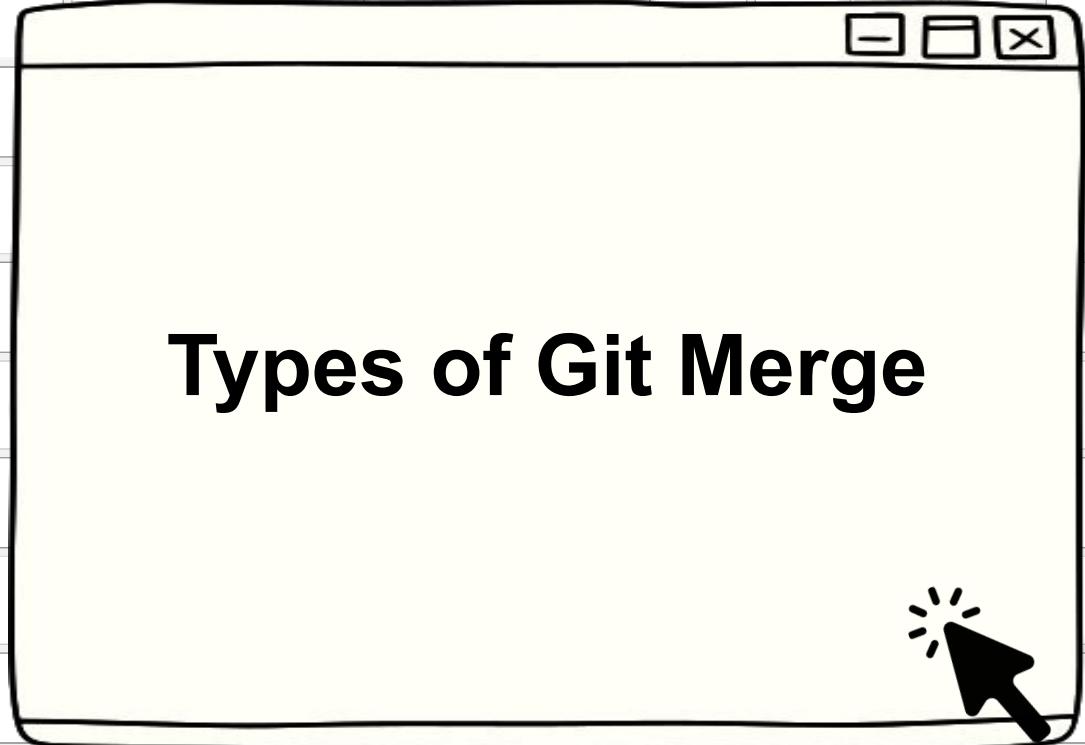
Karan Gupta

## Basic Merge Workflow:

- ① Switch to the target branch (`main` or `develop`).
- ② Use `git merge <branch>` to integrate changes.
- ③ Resolve merge conflicts if needed.
- ④ Push merged changes to the remote repository.

## Best Practices:

- **Merge Frequently:** Minimize conflicts by integrating changes regularly.
- **Review Changes:** Use `git diff` to review differences before merging.
- **Resolve Conflicts Carefully:** Ensure all conflicts are resolved correctly.
- **Test After Merging:** Verify functionality after the merge.

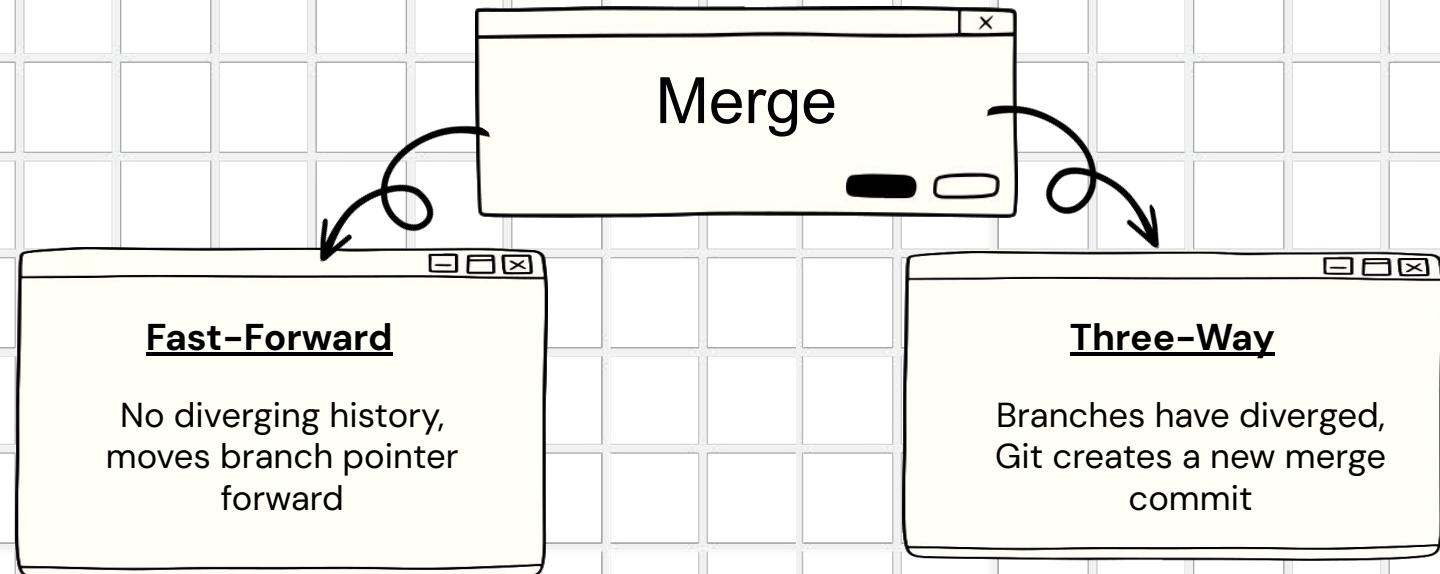




Karan Gupta



# Types of Git Merge

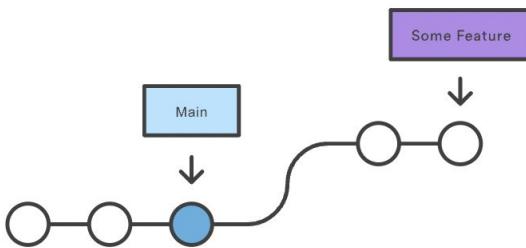




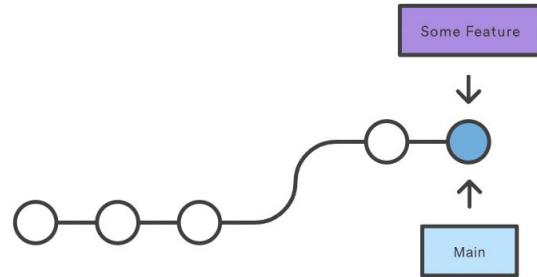
Karan Gupta



Before Merging



After a Fast-Forward Merge

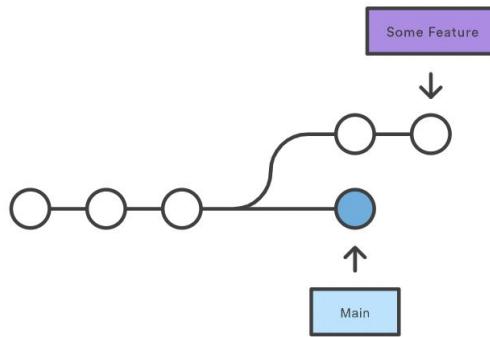




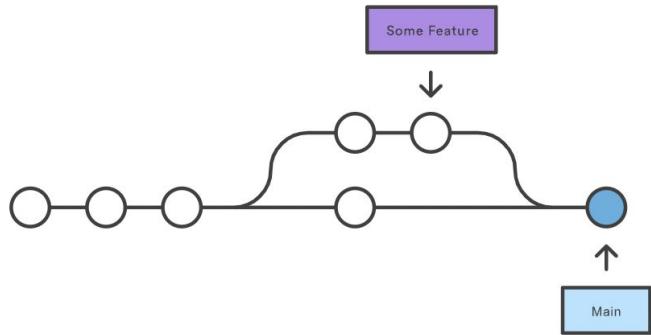
Karan Gupta



Before Merging



After a 3-way Merge



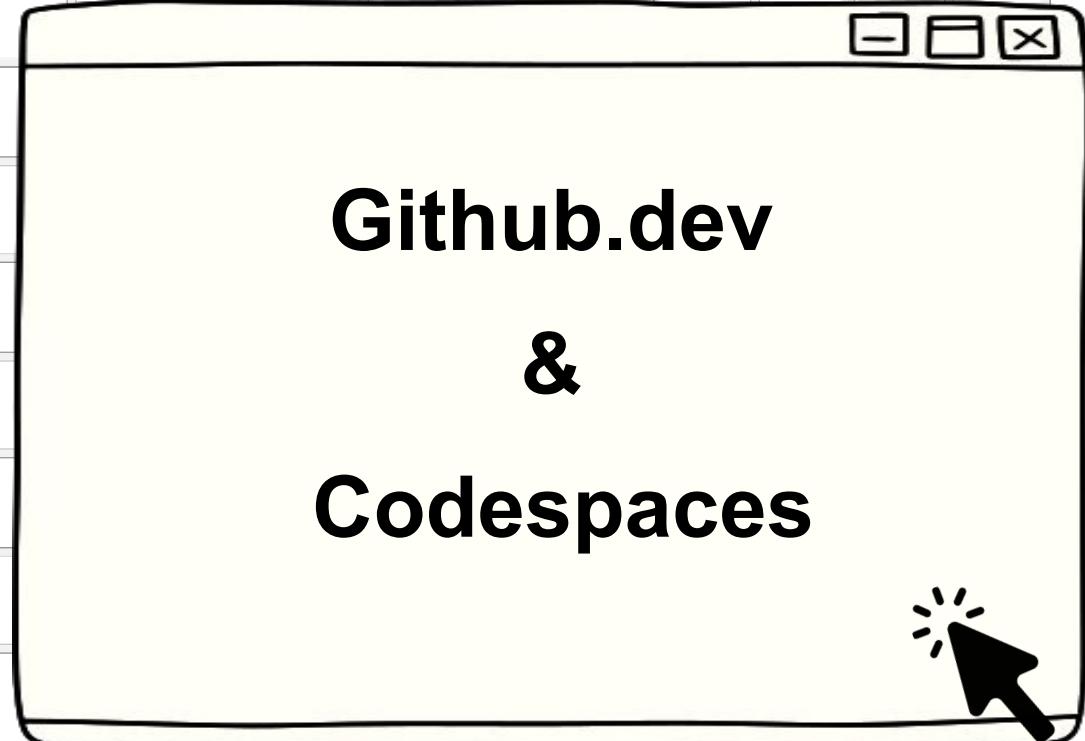
# Three-Way Merge Vs Fast-Forward Merge



Karan Gupta



Feature	Three-Way Merge	Fast-Forward Merge
<b>When It Happens</b>	When both branches have new commits.	When the target branch has no new commits since branching off.
<b>How It Works</b>	Git creates a <b>new merge commit</b> combining changes.	Git <b>moves the branch pointer forward</b> without creating a merge commit.
<b>Commit History</b>	Preserves a clear history of merges.	Linear history, as if all changes were made on the main branch.
<b>Merge Conflicts</b>	Possible, needs manual resolution.	No conflicts (unless changes exist in the same file before merging).



# Github.dev



Karan Gupta

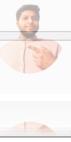
- A **lightweight, browser-based editor** for GitHub repositories.
- Accessible via [github.dev](https://github.dev) or by pressing `.` in any GitHub repository.
- Uses **Visual Studio Code (VS Code) Web** for a familiar experience.

## Key Features:

- ✓ **Instant Editing** – No setup required, edit files directly in the browser.
- ✓ **VS Code Experience** – Syntax highlighting, extensions, and themes.
- ✓ **Live Preview** – Works well for static files like HTML, Markdown, and JSON.
- ✓ **No Terminal or Extensions** – Unlike VS Code Desktop, [github.dev](https://github.dev) has **limited functionality** (no terminal, debugging, or running code).

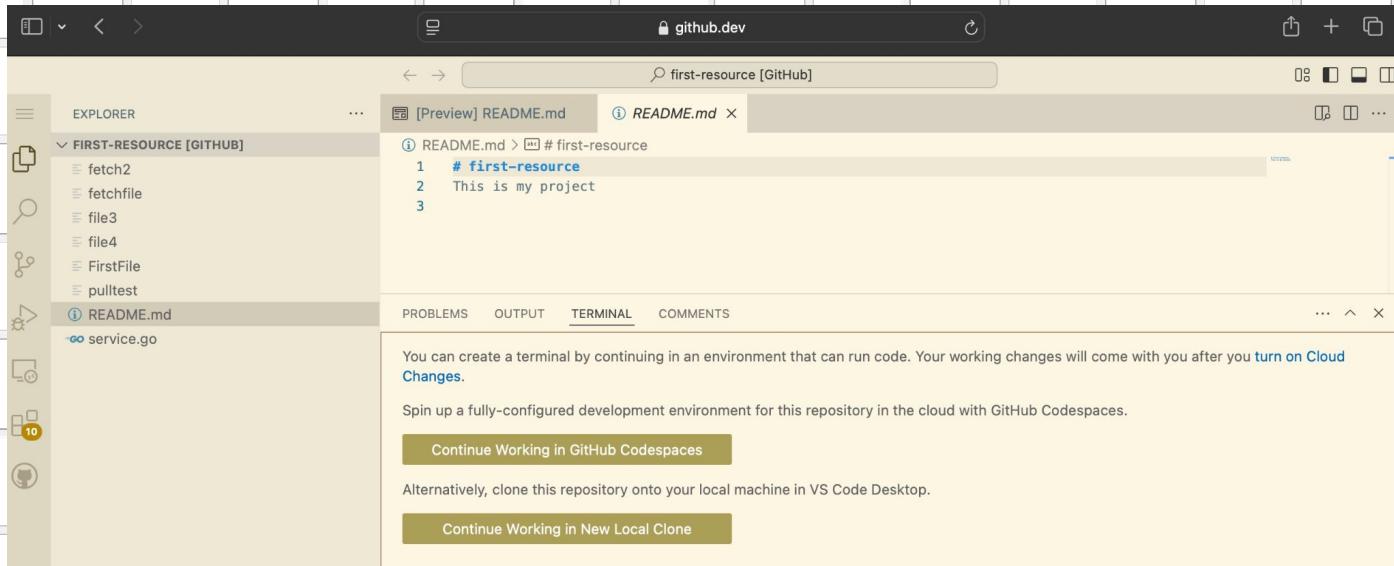


Karan Gupta



## When to Use GitHub.dev?

- Quick edits on GitHub repositories.
- Reviewing & making small changes in code.
- Editing markdown files, README, and documentation.





Karan Gupta

# Codespaces

- A **cloud-based development environment** powered by **VS Code**.
- Provides **fully-configurable Linux-based virtual machines** with pre-installed tools.
- Supports **running, debugging, and testing code in the cloud**.

## Key Features:

- ✓ **Pre-configured Dev Environments** – Start coding instantly without local setup.
- ✓ **Terminal & Debugging** – Unlike [github.dev](#), Codespaces supports **full VS Code features**.
- ✓ **Customizable VM** – Choose CPU, memory, and pre-installed dependencies.
- ✓ **Works on Any Device** – Access your dev environment from anywhere.

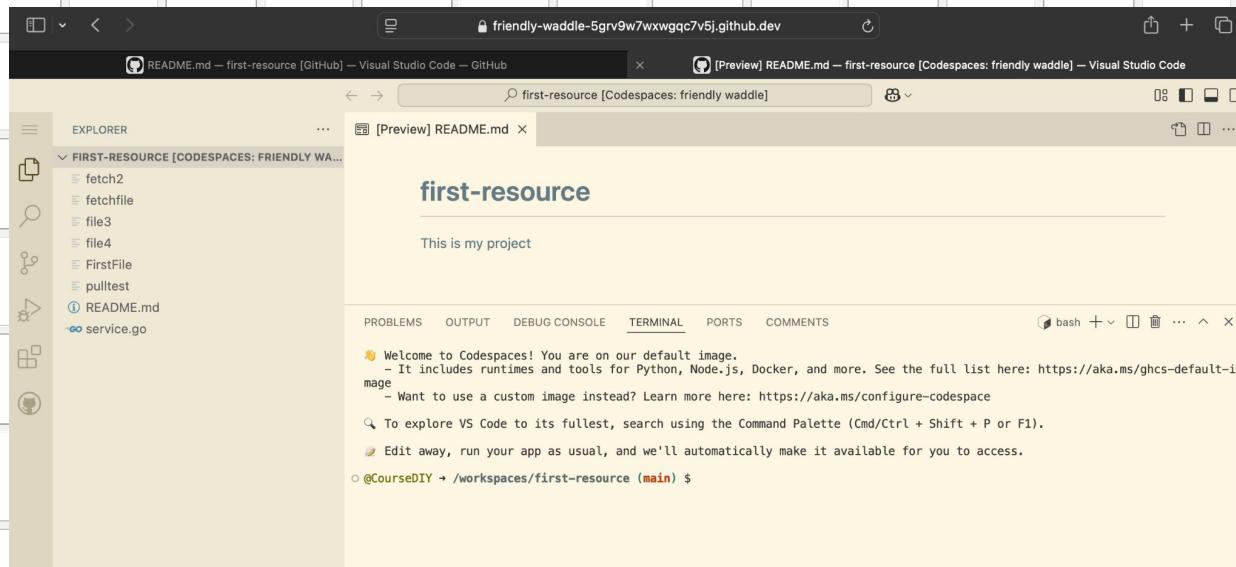


Karan Gupta



## When to Use Codespaces?

- Collaborative development & onboarding new developers.
- Working on large projects without setting up a local machine.
- Running, testing, and debugging applications in the cloud.







# Git Rebase

- `git rebase` moves commits from one branch **on top of another branch**.
- It helps maintain a **clean, linear commit history** by avoiding unnecessary merge commits.
- Often used to **integrate changes** from the main branch into a feature branch.

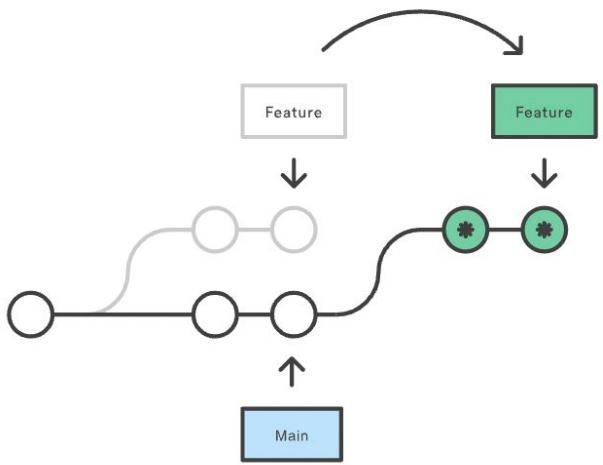
## Purpose:

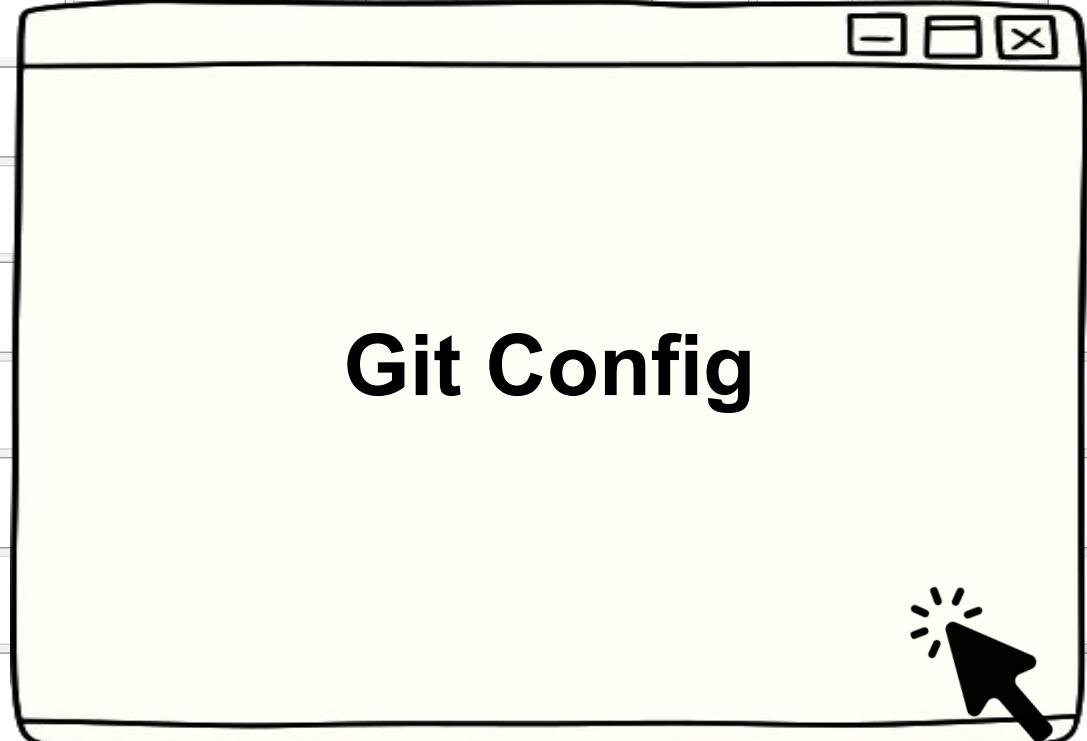
- **Linear History:** Simplifies project history.
- **Update Feature Branches:** Integrate changes from main without merge commits.
- **Resolve Conflicts Early:** Address conflicts during rebase.

## Best Practices:



- **Use Rebase for Local Changes:** Rebase local changes before pushing.
- **Avoid Rebasing Shared Branches:** Prevent complications with shared branches.
- **Keep Commits Clean:** Use interactive rebase for clean commit history.
- **Test Thoroughly:** Run tests after rebasing.







# Git Config

## ◆ Why Configure Git?

- ❖ Ensures correct identity for commits.
- ❖ Improves workflow with aliases and default settings.
- ❖ Enables seamless integration with remote repositories.

## ◆ Example:

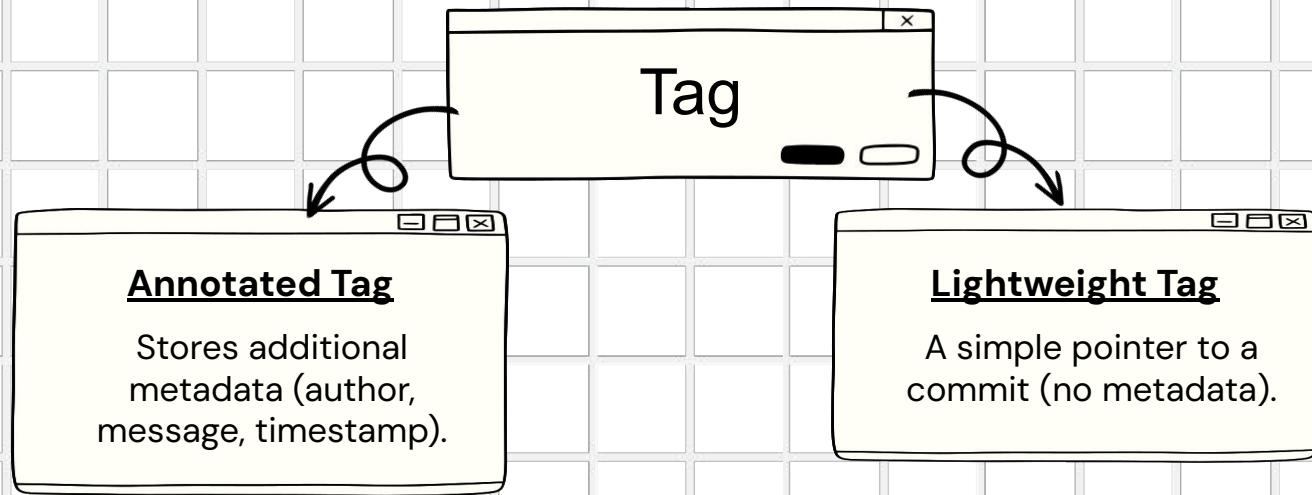
- `git config --global user.name "Your Name"`
- `git config --global user.email "your.email@example.com"`
- `git config --list`



# Git Tag



- ❖ Git tags are used to **mark specific points** in a repository's history.
- ❖ Often used to tag **releases (v1.0, v2.0, etc.)** for easy reference.
- ❖ Unlike branches, tags are **immutable** and do not change over time.







Karan Gupta

# Git Blame

- ❖ `git blame` helps track **who last modified each line** of a file and when.
  - ❖ Useful for debugging and understanding changes in the codebase.
  - ❖ Shows the **commit hash, author, and timestamp** for each line.
- 
- ◆ Example:

```
git blame filename
```





# Git Clean



Karan Gupta

- ❖ `git clean` removes **untracked files** from your working directory.
- ❖ Helps keep the repository clean by deleting unwanted files.
- ❖ Does **not** affect tracked files or staged changes.

## When to Use `git clean`?

- When your working directory has **untracked temporary files**.
  - To remove **build artifacts, log files, or ignored files**.
  - Before switching branches to avoid **file conflicts**.
- ♦ Example:

`git clean`



# Git Reset



- Undo changes by resetting the current branch to a specified state.
- Modifies the staging area and optionally the working directory.

## Types of Resets:

- **Soft Reset:**
  - `git reset --soft <commit>`
  - Moves HEAD, keeps changes staged.
- **Mixed Reset (default):**
  - `git reset --mixed <commit>` or `git reset <commit>`
  - Moves HEAD, resets index, keeps working directory.
- **Hard Reset:**
  - `git reset --hard <commit>`
  - Moves HEAD, resets index and working directory, discards changes.



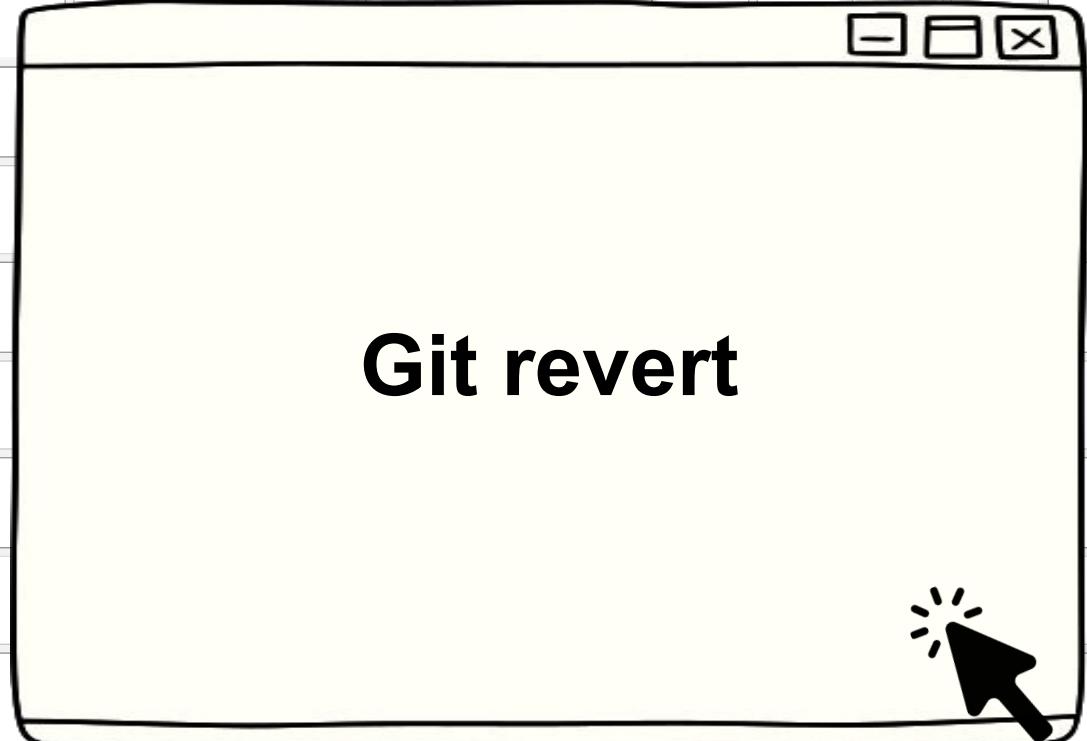
Karan Gupta



## Best Practices:

- **Use Soft and Mixed Reset for Safe Changes:** Undo commits or unstage changes without losing work.
- **Avoid Hard Reset on Shared Branches:** Can disrupt collaborators; use with caution.
- **Be Cautious with Hard Reset:** Discards changes irreversibly; ensure you have backups.
- **Review Changes Before Resetting:** Use `git status` and `git diff`.







Karan Gupta

# Git Revert

- Creates a new commit that undoes the changes introduced by a previous commit.
- Safe way to undo changes while preserving commit history.

## Purpose:

- **Undo Changes Safely:** Revert commits without altering history.
- **Maintain History:** Clear record of what was undone.
- **Fix Mistakes:** Correct specific issues while keeping other changes.

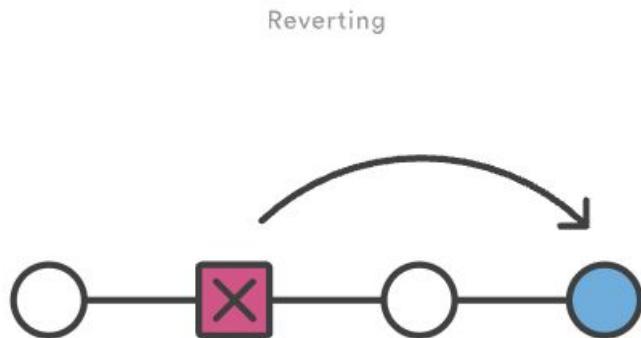
## Command Usage:

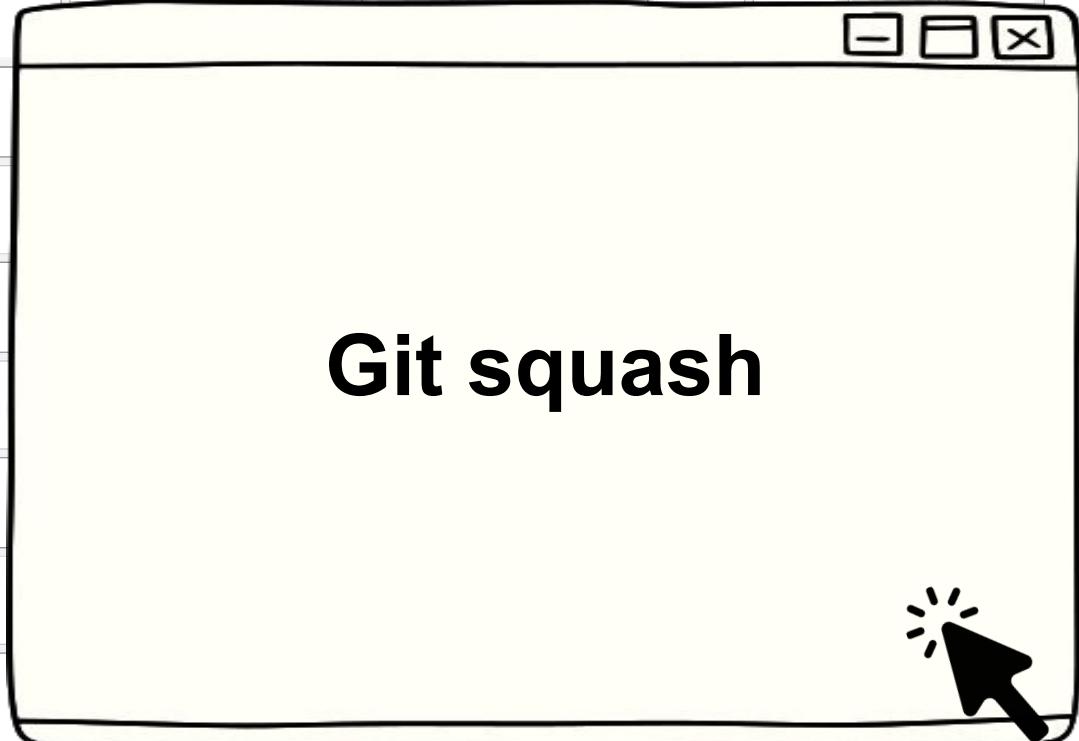
- `git revert <commit>`



## Best Practices:

- **Revert Instead of Reset for Shared Branches:** Avoid disrupting shared history.
- **Test After Reverting:** Run tests to ensure functionality.
- **Review Changes Before Reverting:** Use `git log` and `git diff`.







# Git Squash

## Overview:

- **Squashing Commits:** Combine multiple commits into one for a cleaner history.
- **Purpose:** Clean up commit history, make it more readable before merging.

## When to Use:

- **Feature Development:** Consolidate related commits.
- **Code Review:** Present organized commit history.
- **Bug Fixes:** Streamline history with combined updates.

## Command Usage:

- `git rebase -i <base-commit>`





Karan Gupta

## Overview:

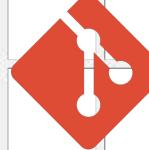
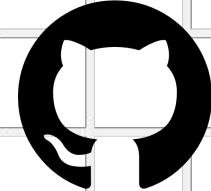
- **Git Cherry-Pick:** Apply specific commits from one branch to the current branch.
- **Purpose:** Integrate selective changes without merging the whole branch.

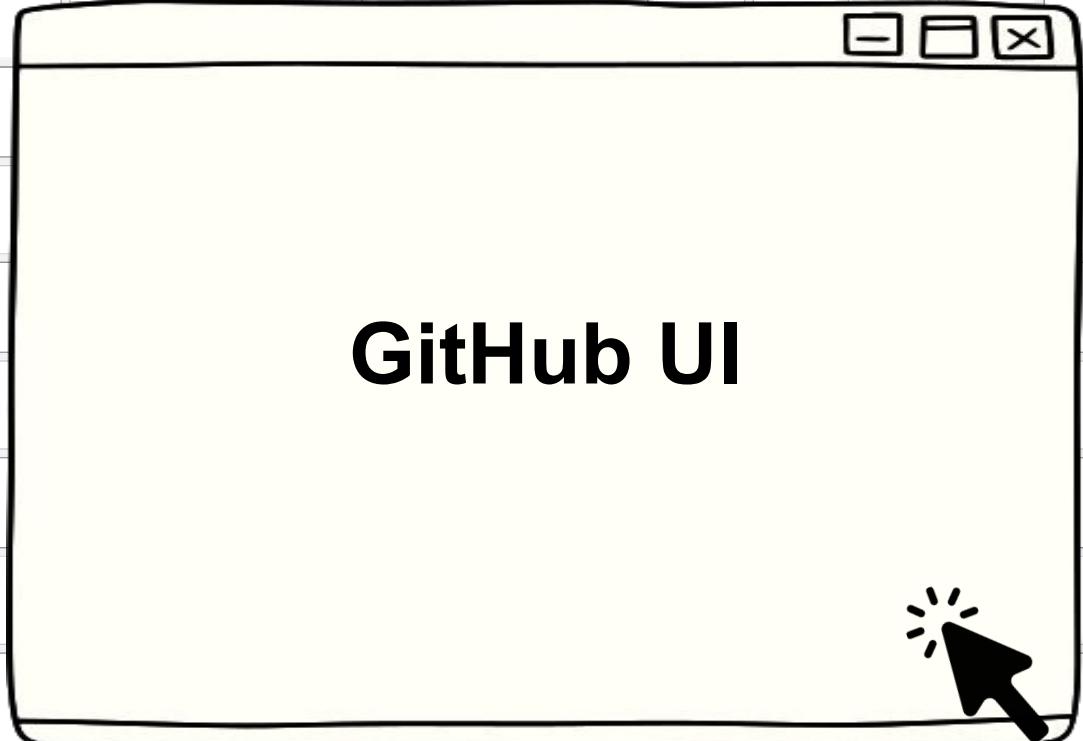
## When to Use:

- **Selective Integration:** Apply specific changes from another branch.
- **Bug Fixes:** Move specific fixes to another branch.
- **Feature Application:** Include specific changes into your branch.

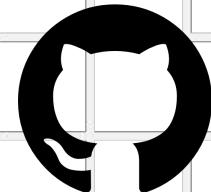
## Command Usage:

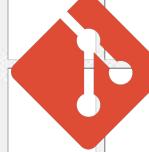
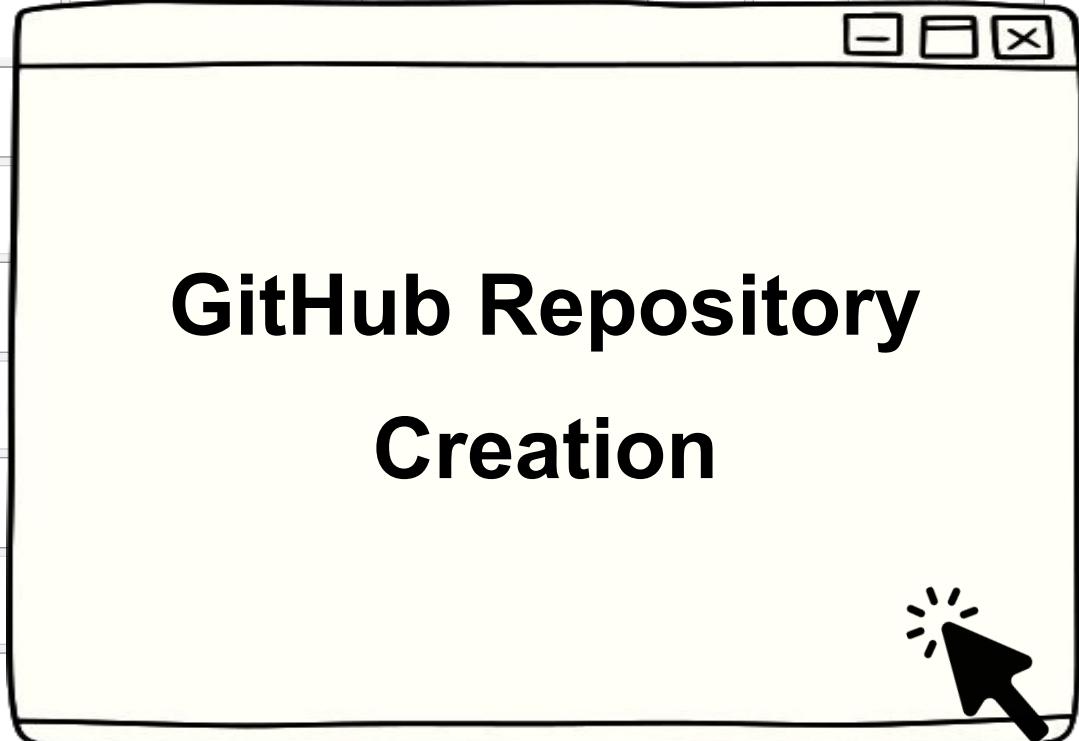
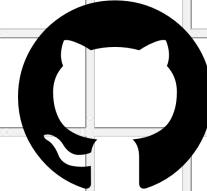
- `git cherry-pick <commit>`

 git



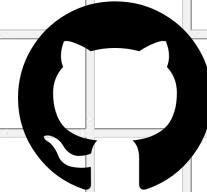
git

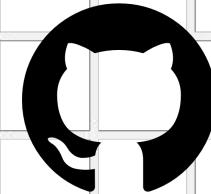
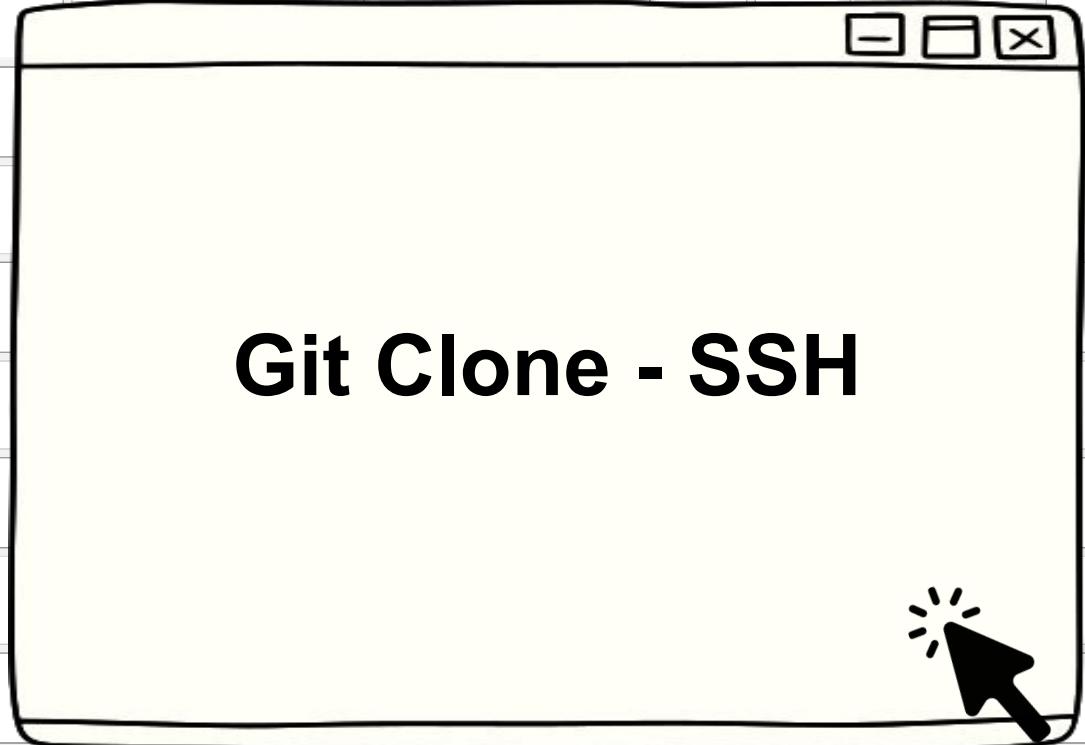


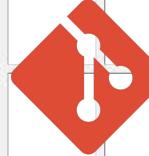
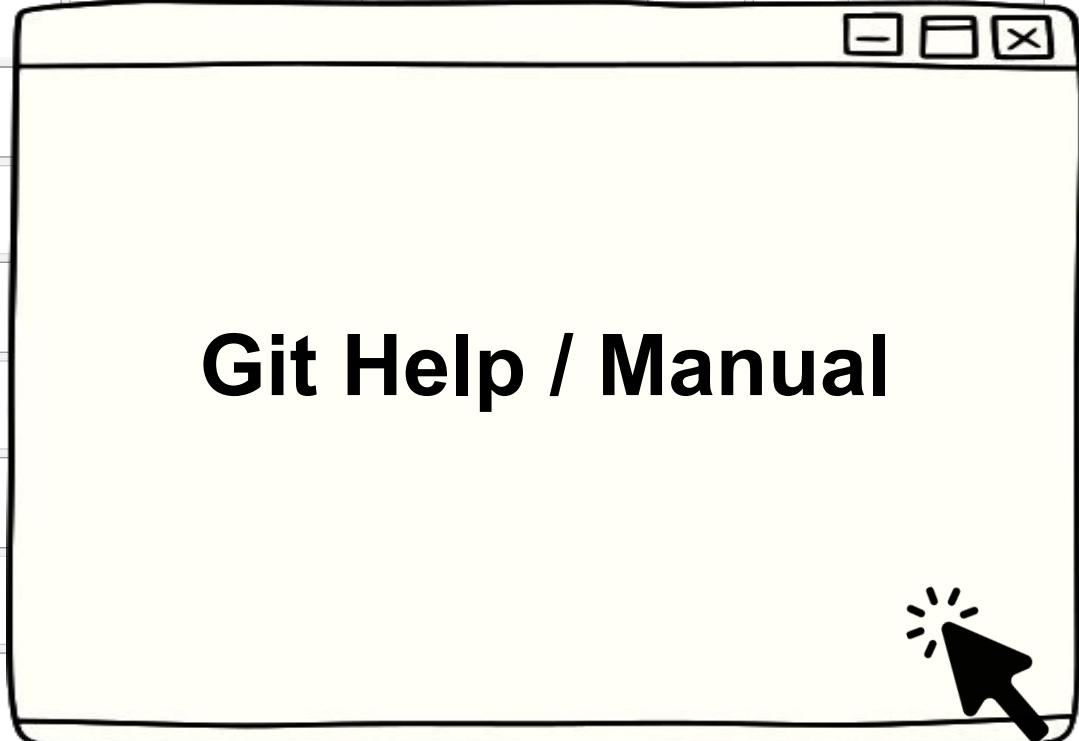
 git



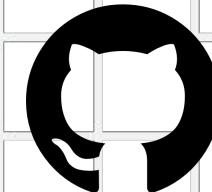
# Creating Files in GitHub Repository







git

A red diamond-shaped icon containing a white git logo symbol (a white line forming a hexagon with two lines extending from the center). To the right of the icon, the word "git" is written in a large, bold, black sans-serif font.



Karan Gupta



## Git Help:

- Provides built-in documentation for Git commands and concepts.
- Access via `git help <command>` or `<command> --help`.
- Examples:
  - `git help commit`
  - `git pull --help`

## man (Manual) Pages:

- View detailed instructions about Git commands.
- Equivalent to `git help`.
- Syntax: `man git-<command>`.
- Example: `man git-merge`.

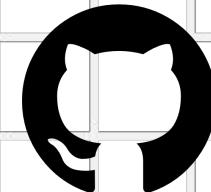


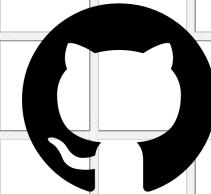
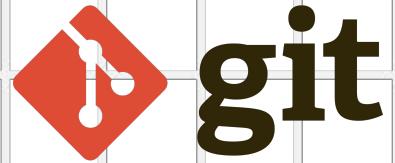


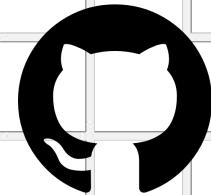
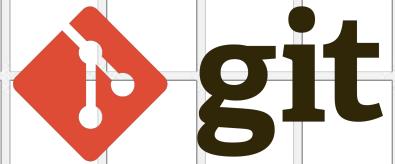
## Command Options:

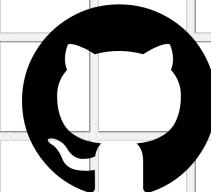
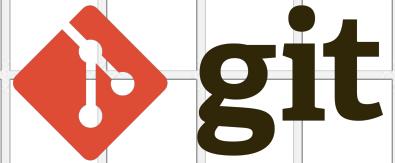
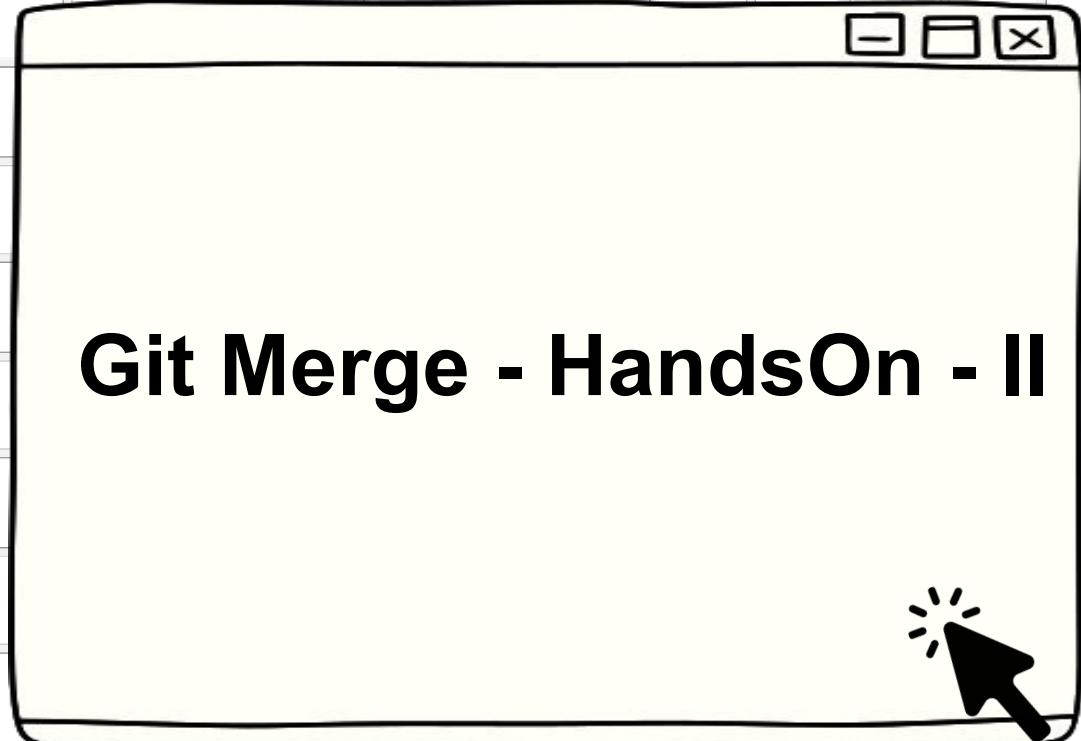
- Understand options and flags:
  - `git log --help` explains log customization.
  - `git status --help` lists useful options.

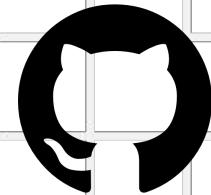
```
karangupta@KaranGupta first-resource % git help
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--no-lazy-fetch]
           [--no-optional-locks] [--no-advice] [--bare] [--git-dir=<path>]
           [--work-tree=<path>] [--namespace=<name>] [--config-env=<name>=<envvar>]
           <command> [<args>]
```













# Git Rebase Interactive Reword



# Git Rebase Reword



## What is Reword in Git Rebase?

- **reword** is used in **interactive rebase** to **change a commit message**
- Keeps the **commit content exactly the same**
- Ideal for fixing typos or clarifying commit messages

## Why Use It?

- Fix typos in messages
- Improve commit clarity for reviewers
- Standardize message format (e.g., JIRA IDs, etc.)



# Git Restore



## What is Git Restore?

- A command to discard changes and revert files to the last committed state.

## Why Use It?

- To undo unwanted modifications before committing.
- To unstage a file from the staging area.
- To recover deleted files before committing.

## Practical Usage of Git Restore

- **Discard Uncommitted Changes** – Reverts file to last committed state.

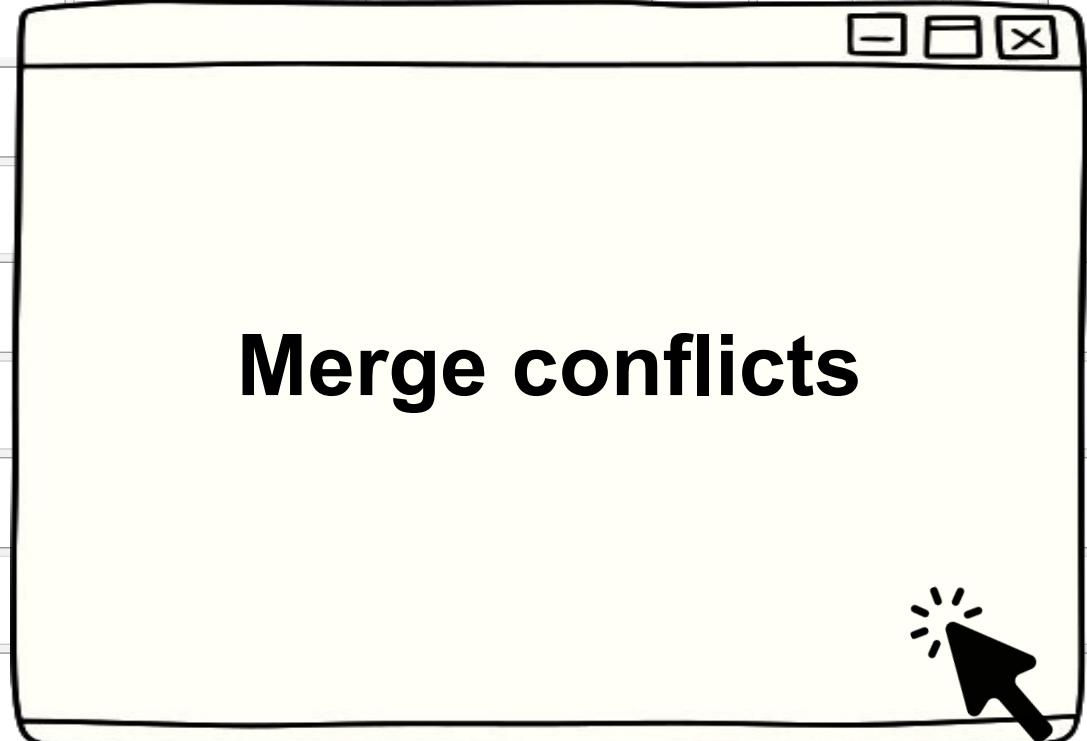
```
git restore file.txt
```

- **Unstage a File (Keep Changes in Working Directory)** – Moves file from staging back to modified state.

```
git restore --staged file.txt
```

- **Restore a Deleted File** – Brings back a deleted file.

```
git restore file.txt
```



# Merge Conflict



## What is Merge Conflict?

- A **merge conflict** occurs when Git cannot automatically merge changes from different branches.
- This happens when **two branches modify the same file in different ways**.
- Common Causes of Merge Conflicts
  - Editing the same lines in a file on different branches.
  - Deleting a file in one branch but modifying it in another.
  - Renaming a file in one branch but keeping the old name in another.



Karan Gupta



## Resolving Conflict:

- 1 Trigger a Merge Conflict. If a conflict occurs, Git shows:

CONFLICT (content): Merge conflict in file.txt

- 2 Open the Conflicted File

```
<<<<< HEAD  
Changes from main branch  
=====  
Changes from feature-branch  
>>>>> feature-branch
```

- 3 Manually Resolve the Conflict

- 4 Mark the Conflict as Resolved





# Merge Conflict



## What is Merge Conflict?

- A **merge conflict** occurs when Git cannot automatically merge changes from different branches.
- This happens when **two branches modify the same file in different ways**.
- Common Causes of Merge Conflicts

- Editing the same lines in a file on different branches.
- Deleting a file in one branch but modifying it in another.
- Renaming a file in one branch but keeping the old name in another.



Karan Gupta

## Resolving Conflict:

- 1 Trigger a Merge Conflict. If a conflict occurs, Git shows:

CONFLICT (content): Merge conflict in file.txt

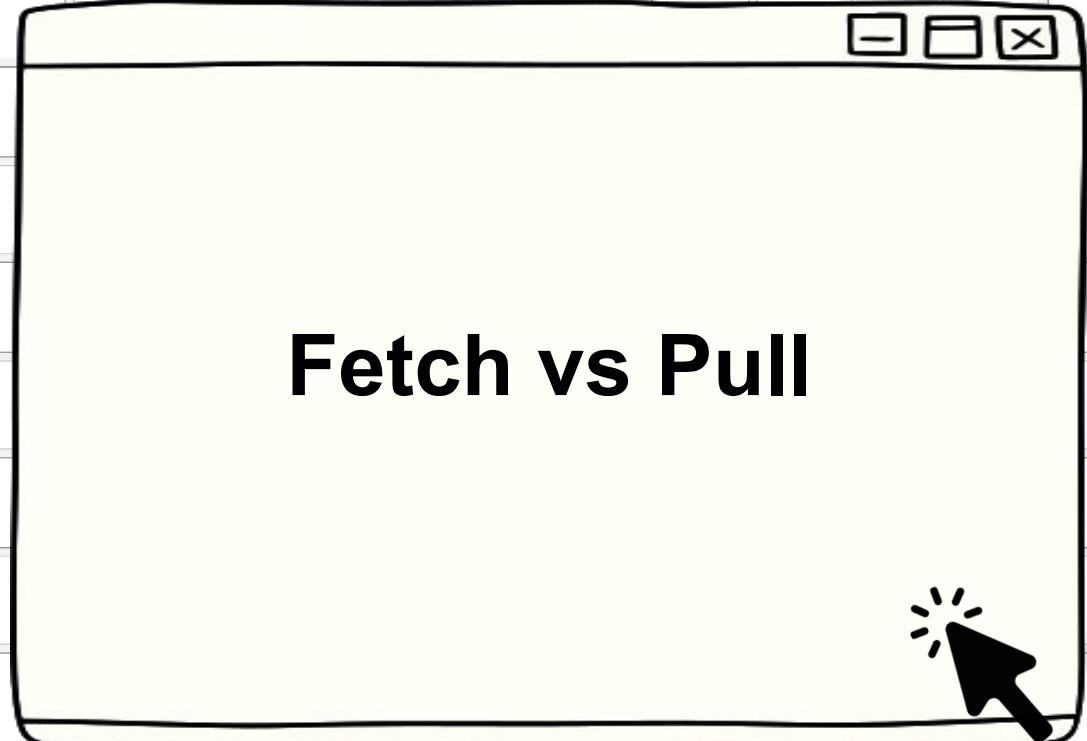
- 2 Open the Conflicted File

```
<<<<< HEAD
Changes from main branch
=====
Changes from feature-branch
>>>>> feature-branch
```

- 3 Manually Resolve the Conflict

- 4 Mark the Conflict as Resolved





# Fetch vs Pull



- **Git Fetch**

- Downloads **remote changes** but does **not** update local branches.
- Lets you review updates **before merging them**.
- Safe—won't modify your working directory.

- **Git Pull**

- Downloads **and merges** remote changes into your local branch.
- Directly updates your working directory.
- May cause conflicts if local changes differ from remote changes.



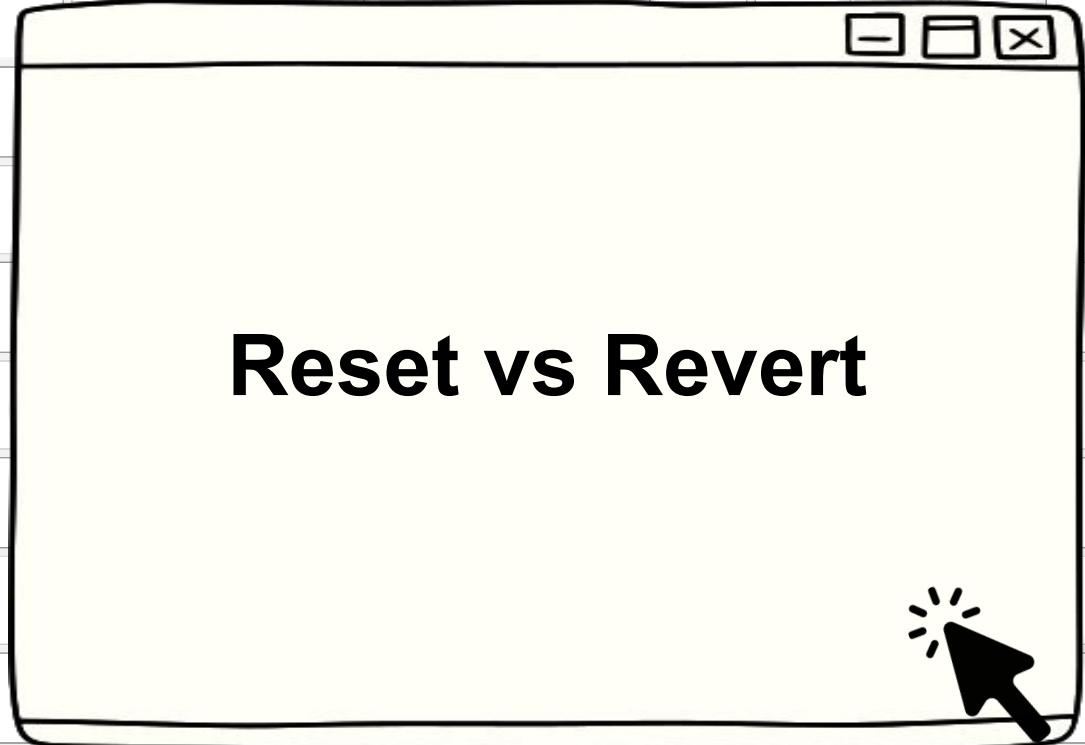
Karan Gupta



Feature	git fetch	git pull
Action	Only downloads changes	Downloads and merges changes
Working Directory	Stays unchanged	Updated immediately
Merge Needed?	Yes, done manually	Automatic merge happens
Use Case	Reviewing changes before merging	Directly updating your branch

✓ Use **fetch** when you want to **check changes first** before merging.

✓ Use **pull** when you want to **update your branch immediately**.



# Reset vs Revert



- **Git Reset** – Moves the branch pointer to a previous commit

## Types:

- `--soft` → Keeps changes staged
- `--mixed` → Unstages changes (default)
- `--hard` → Deletes changes permanently

- **Git Revert** –

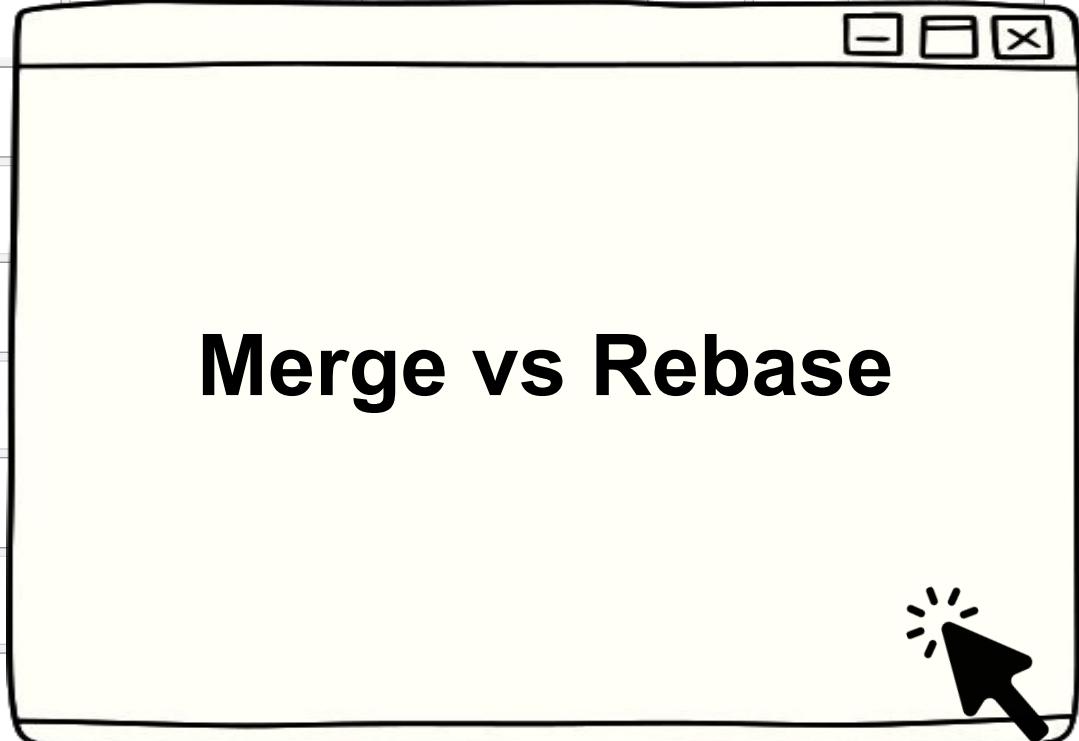
- Creates a new commit that **undoes** a specific commit
- Preserves history (safer than reset)
- Does not modify past commits, making it great for team collaboration



Feature	git reset	git revert
<b>Changes history?</b>	Yes (moves branch pointer)	No (creates a new commit)
<b>Modifies past commits?</b>	Yes	No
<b>Safe for teamwork?</b>	✗ No	✓ Yes
<b>Removes commit from history?</b>	✓ Yes	✗ No
<b>Undo last commit?</b>	<code>git reset HEAD~1</code>	<code>git revert HEAD~1</code>



⚠ Warning: `git reset --hard` erases commits!



# Merge vs Rebase



- **Git Merge**

- Combines branches, keeping commit history
- Creates a **merge commit**
- Keeps the original branch structure intact

- **Git Rebase**

- Moves commits on top of another branch
- **Rewrites commit history**
- Creates a **linear commit history**



Feature	git merge	git rebase
Commit History	Preserves full history with a merge commit	Creates a linear history
New Commit?	✓ Yes (merge commit)	✗ No (rewrites commits)
Best For	Preserving history (team projects)	Clean, linear history (solo work)
Safer?	✓ Yes (doesn't modify past commits)	⚠ No (alters history, requires force push)

- ✓ Use **git merge** when working in a team to preserve history
- ✓ Use **git rebase** for a clean commit history before merging



Karan Gupta



# Git Cherry-Pick

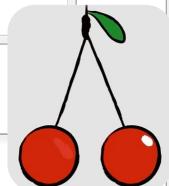


## 🔍 What is git cherry-pick?

- `git cherry-pick` is used to **apply a specific commit** from one branch onto another without merging the full branch.
- You fixed a bug in a feature branch, but need it immediately in `main`—you **cherry-pick** just that fix without merging other in-progress code.

## Features:

- Works great for **hotfixes** or **selective commits**
- Supports cherry-picking **multiple commits**
- If conflict occurs → resolve → `git cherry-pick --continue`





# Git Restore - HandsOn







# Git Revert - HandsOn



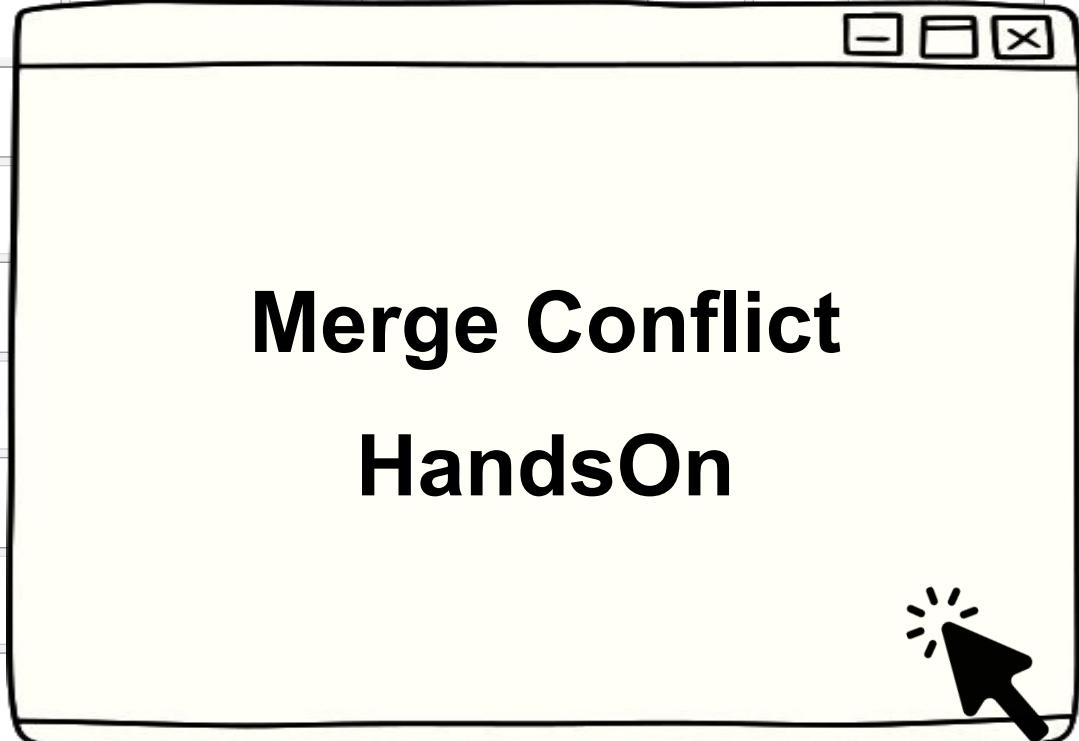


# Git Rebase - HandsOn





Karan Gupta







Karan Gupta

# Git Installation on Mac



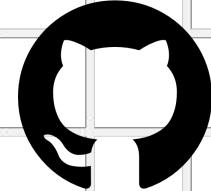


# Git Installation on Windows



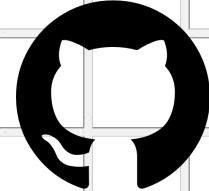


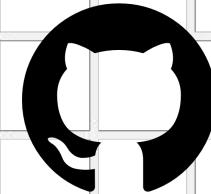
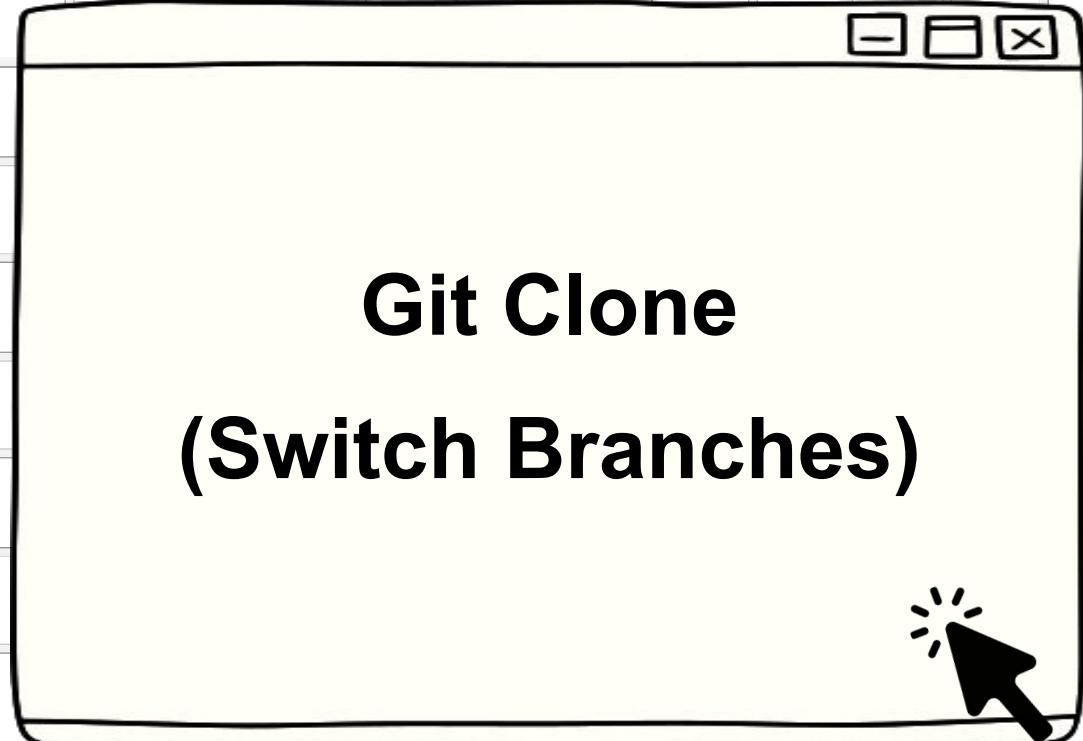
# Creating Branch Directly in GitHub

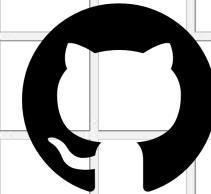
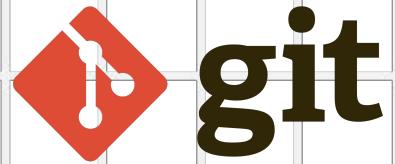




# Committing Local Branch to Remote Repo









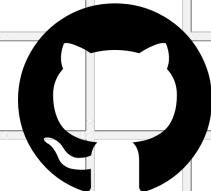


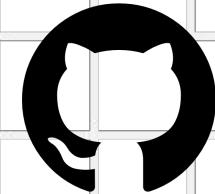
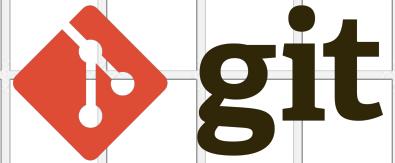
# Pull request - HandsOn

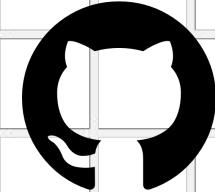


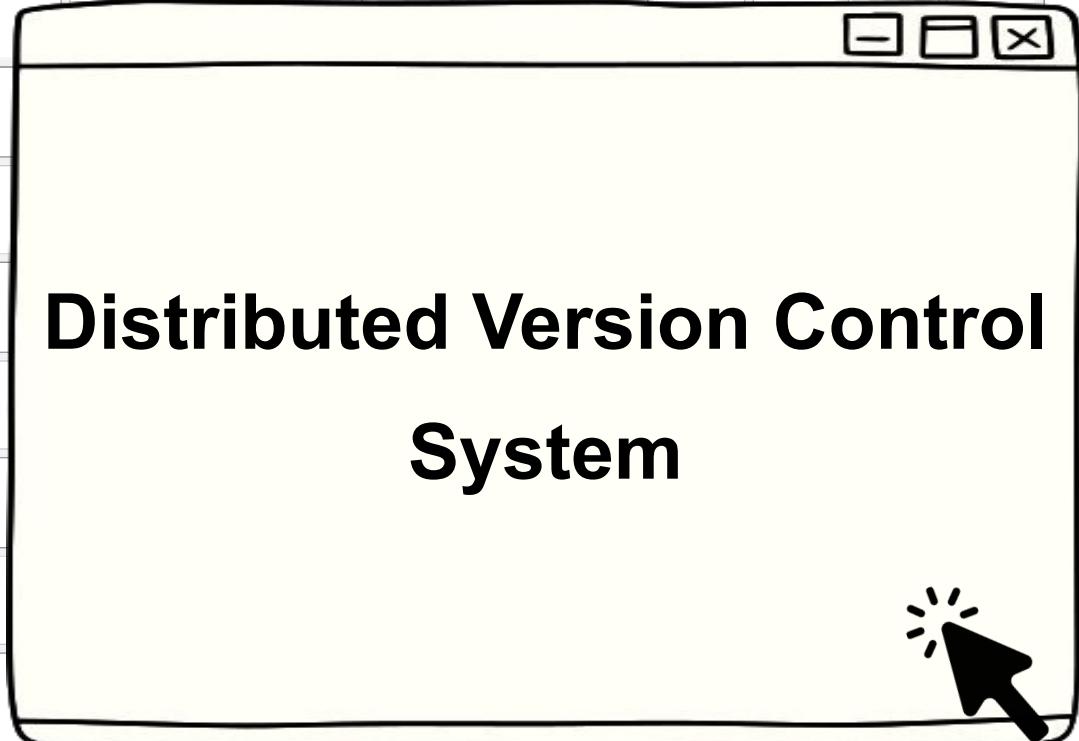


**Real Time Situation:**  
**How to push code from  
local to updated remote  
repo?**









# Distributed Version Control System



## What is DVCS?

A **Distributed Version Control System** allows every contributor to **have a full copy of the repository**, including its complete history.

## Key Characteristics:

- Each user has a **local repository** with full version history.
- Users can commit, view history, and work **offline**.
- Supports **peer-to-peer collaboration**
- Examples: **Git, Mercurial, Bazaar**



## Comparison with Centralized VCS:

Feature	Centralized VCS	Distributed VCS (DVCS)
<b>Repo Location</b>	Central Server	Each User's Machine
<b>Offline Work</b>	✗ No	✓ Yes
<b>Single Point of Failure</b>	✓ Yes	✗ No
<b>Speed &amp; Performance</b>	Slower	Faster



# GitHub Accounts



Account Type	Best For	Features Highlight
Free	Beginners, learners	Public/private repos, community support
Pro	Freelancers	More CI/CD, insights, tools
Team	Dev teams/startups	Collaboration tools, permissions
Enterprise	Large orgs	Advanced security, compliance, enterprise tools

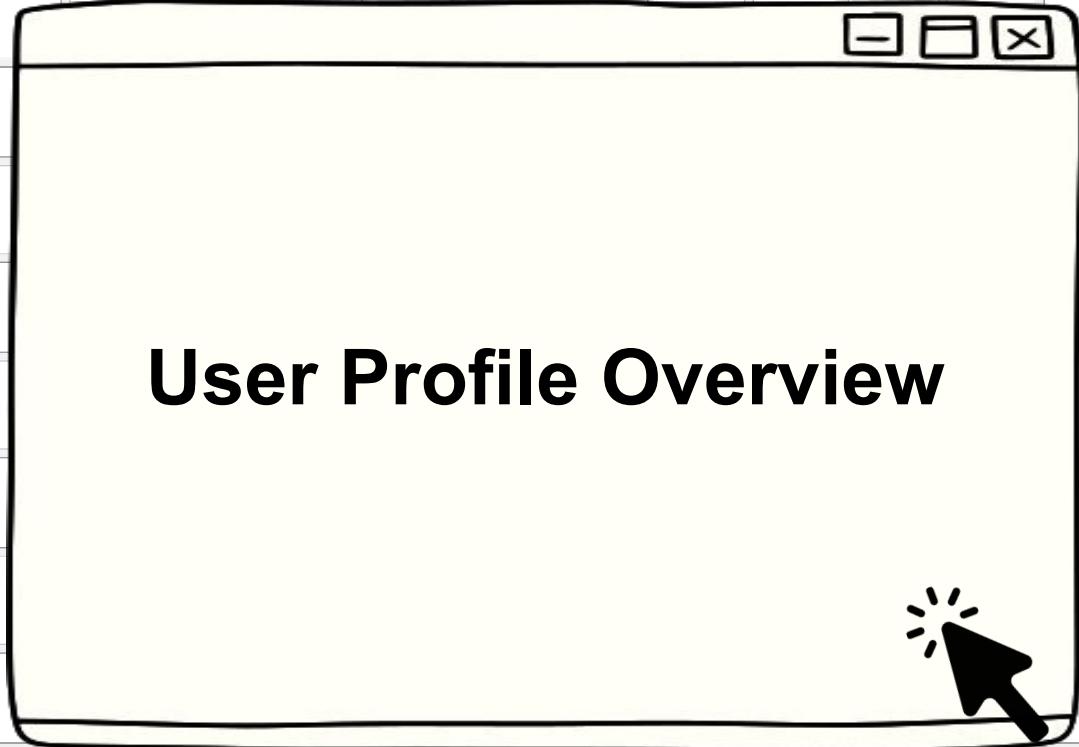


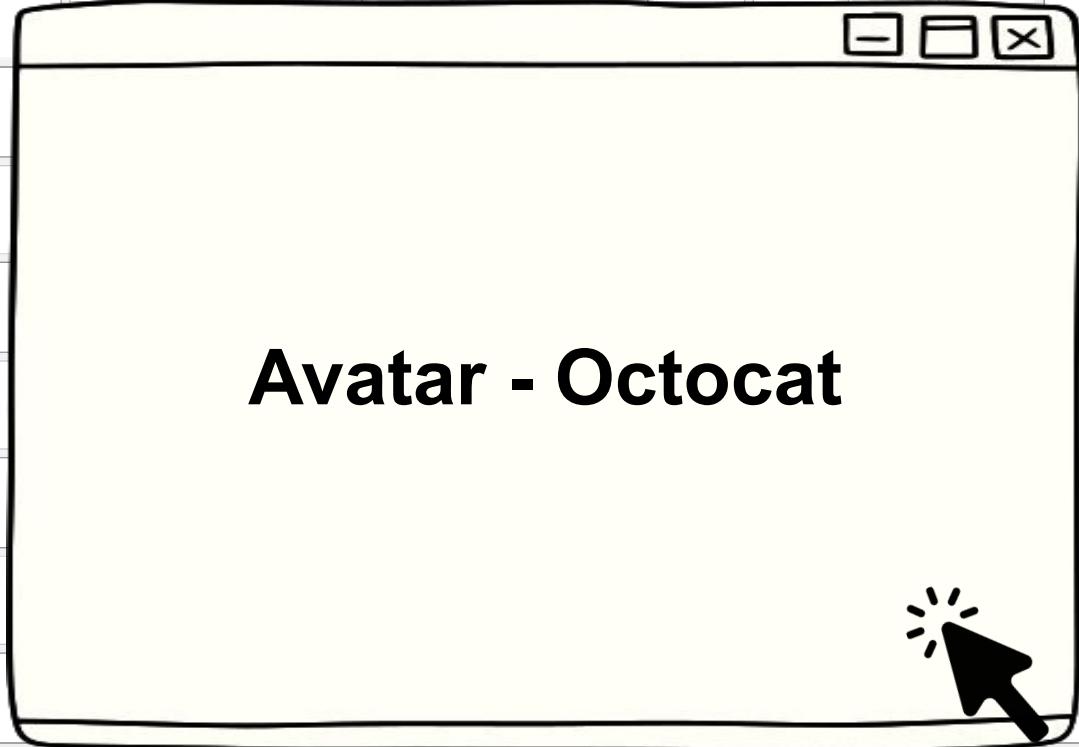
# GitHub Desktop



- GitHub Desktop is a **graphical user interface (GUI)** application that lets you interact with GitHub repositories without using the command line.

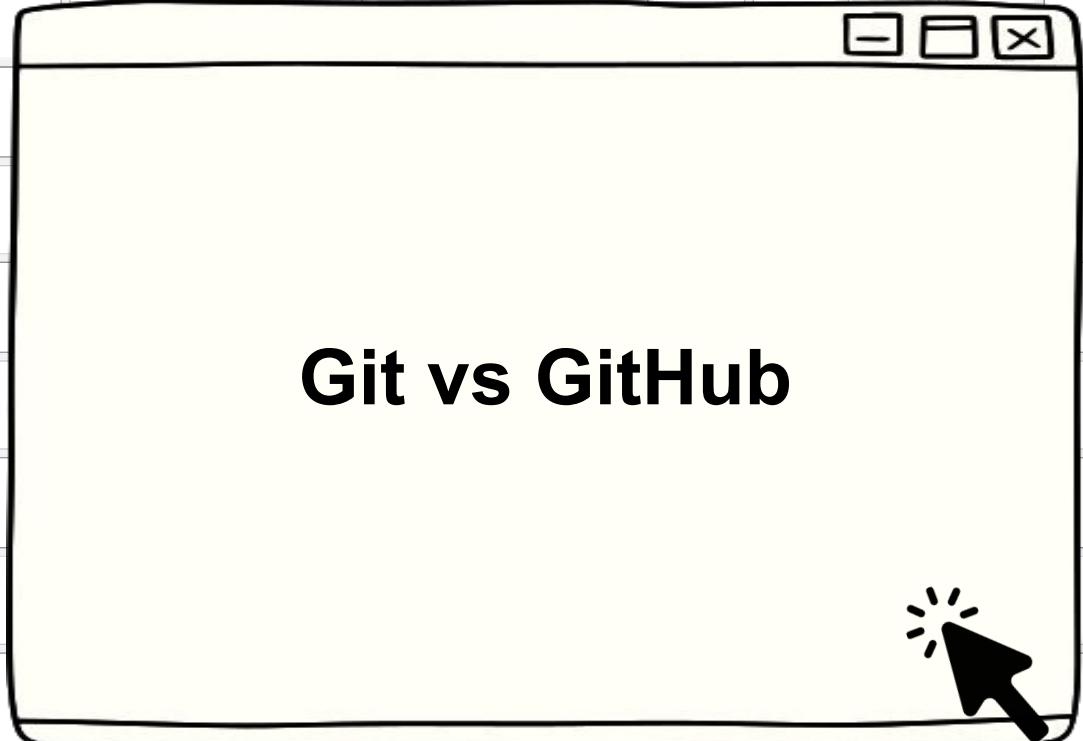
Benefit	Description
Ease of Use	No need to memorize Git commands
Visual Clarity	View changes, commits, and branches clearly
Speed	Perform Git operations faster via UI
Team Friendly	Reduces Git learning curve for collaborators







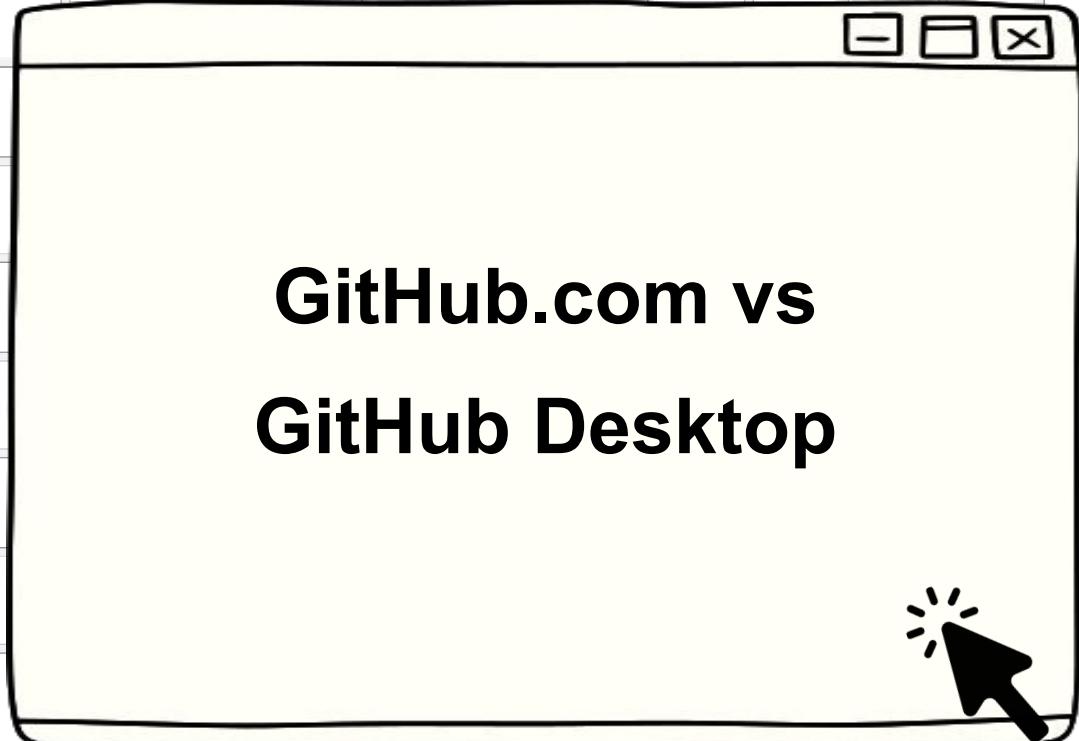
Karan Gupta



# Git vs GitHub

Feature/Aspect	Git 	GitHub 
What is it?	Distributed Version Control System	Git Repository Hosting Platform
Primary Function	Track changes in source code	Store & collaborate on code online
Type	CLI tool installed locally	Cloud-based web service
Developed By	Linus Torvalds (2005)	GitHub, Inc. (2008)
Usage	Local code versioning	Remote repo hosting + collaboration
Works Offline?	 Yes	 No – Requires internet
Features	Commits, branches, merges	Pull requests, issues, CI/CD, team management
Interface	Command line (or GUI clients)	Web interface (also GitHub CLI/Desktop)

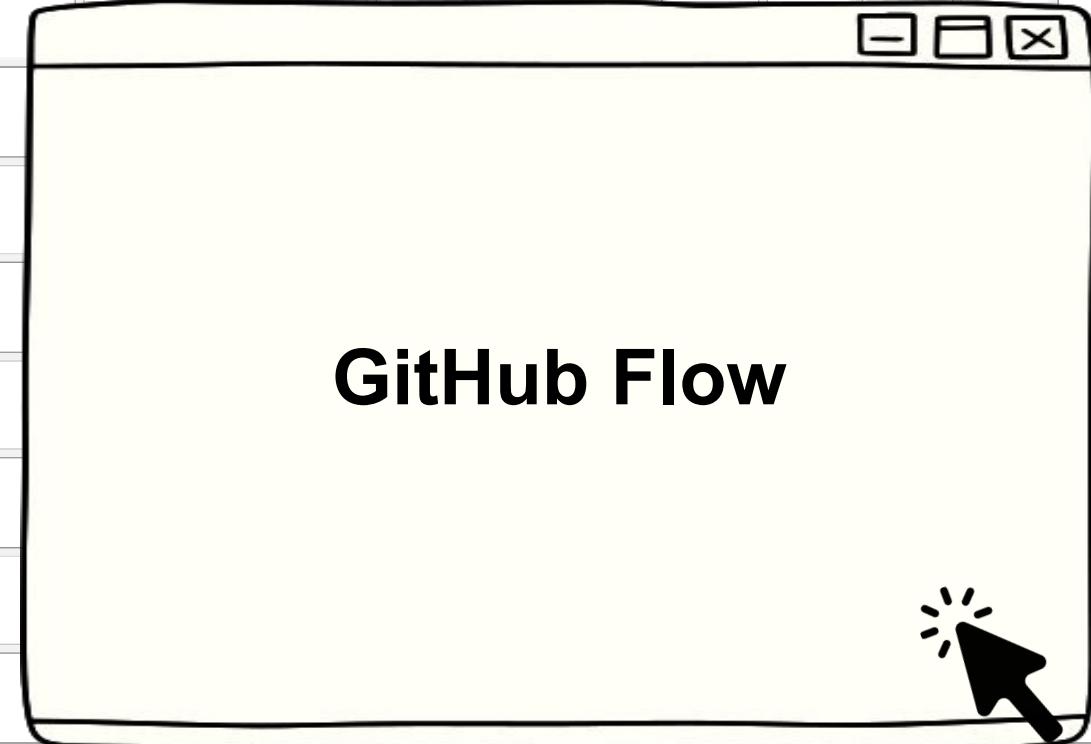




# GitHub.com vs GitHub Desktop



Feature	GitHub.com	GitHub Desktop
Type	Web-based platform	Desktop application
Use Case	Manage repositories, issues, pull requests	Perform Git operations locally with UI
Interface	Web browser	GUI app (Windows, macOS)
Best For	Code hosting, collaboration, project tracking	Beginners or those preferring GUI over CLI
Internet Required	Yes	No (for local work), Yes (to sync with GitHub)



# GitHub Flow

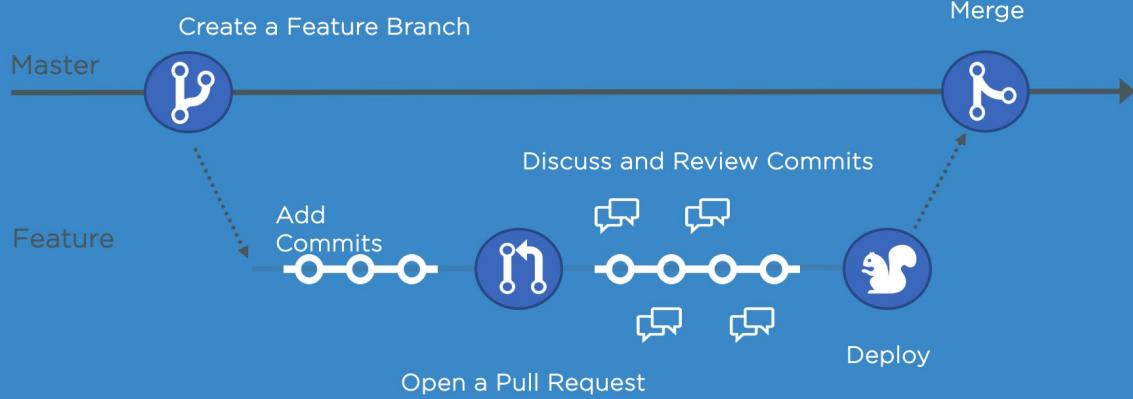


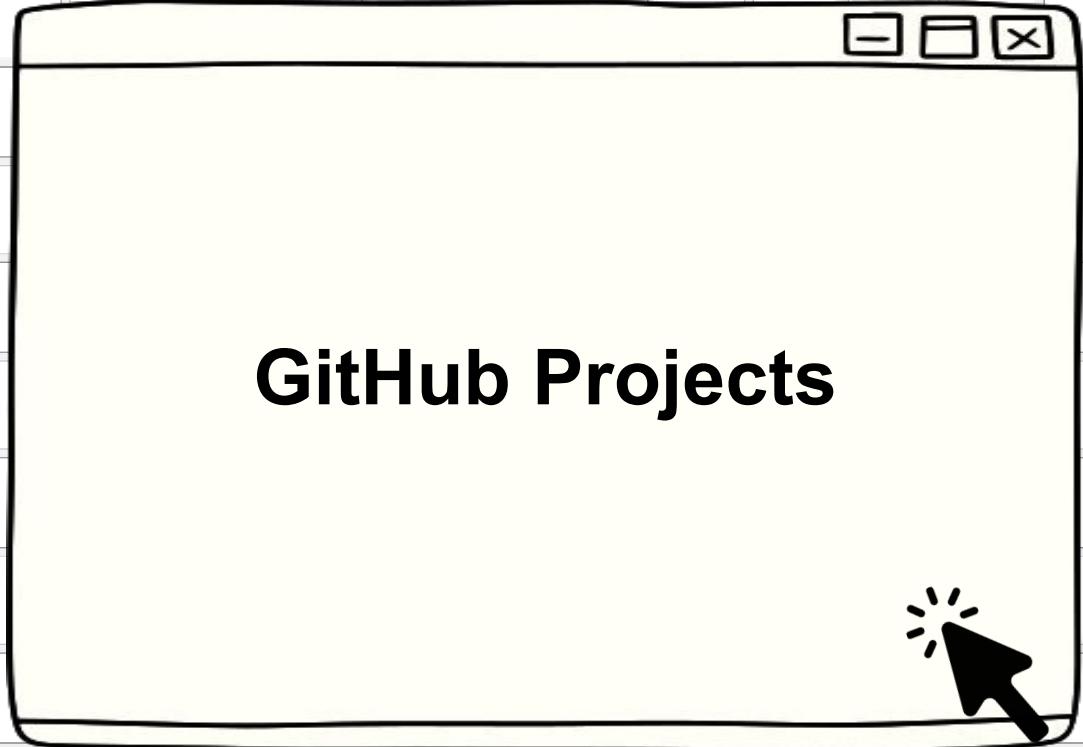
- GitHub Flow is a **lightweight branching strategy** for teams who deploy regularly.
- It supports **continuous delivery** and **collaborative development** using GitHub.

## 7 Core Steps in GitHub Flow:

1. Create a Branch
2. Make Commit
3. Open a Pull Request
4. Review Code
5. Deploy for Testing (Optional)
6. Merge to Main
7. Delete your Branch

# GitHub Flow





# GitHub Projects



- GitHub Projects is a **project management tool** built into GitHub to help teams **organize, plan, and track work** using issues, pull requests, and notes.
- Managing the lifecycle of a software project from feature planning to bug fixing.

## Key Features:

- Kanban-style boards** (like To-Do, In Progress, Done)
- Integration with Issues and Pull Requests**
- Custom workflows** with automation
- Filters, views, and grouping by fields**
- Markdown support** for notes and descriptions



Karan Gupta



## Benefits:

-  **Seamless GitHub Integration:**

Directly links to issues and PRs without needing external tools.

-  **Customizable Workflows:**

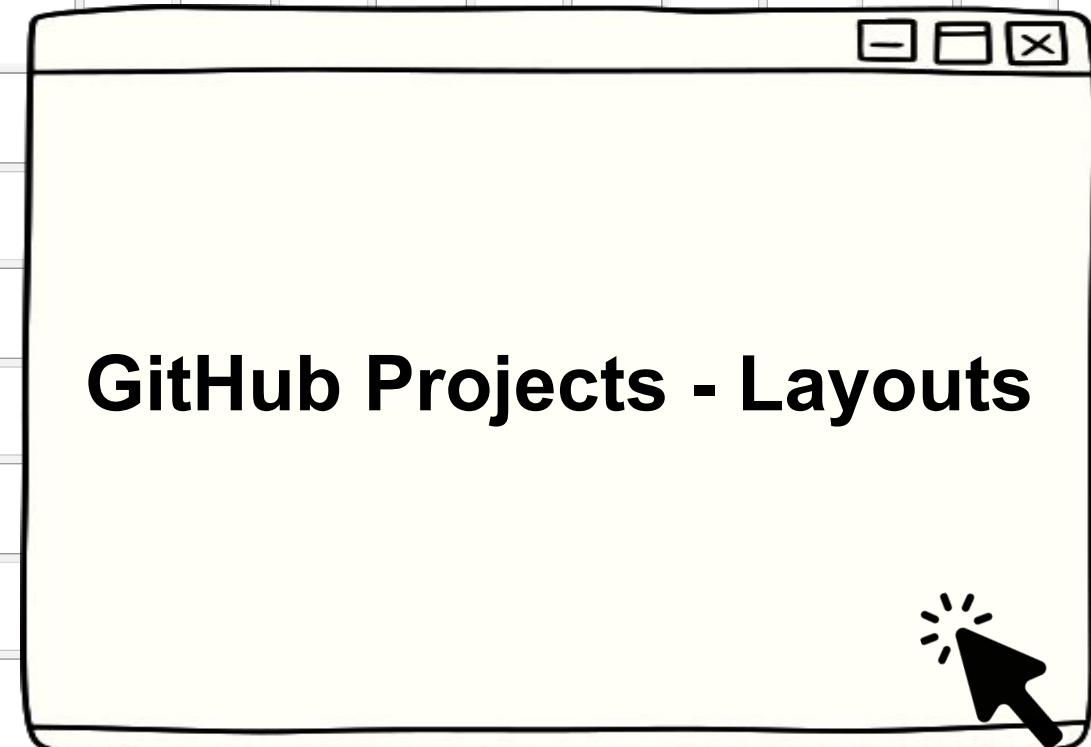
Tailor columns, fields, and views to match your team's needs.

-  **Automation Support:**

Automate task transitions (e.g., move to "Done" on PR merge).

-  **Improved Collaboration:**

All team members stay on the same page with real-time updates.



# GitHub Projects Layouts



## 1. Board View (Kanban Style)

- Classic column-based layout (e.g., To Do → In Progress → Done)
- Best for **agile workflows**, sprint planning, and tracking progress

## 2. Table View

- Spreadsheet-like layout
- Allows sorting, filtering, and editing fields inline
- Best for **bulk editing**, prioritization, or creating custom views



### **3. Roadmap (Timeline View)**

- Visual timeline of work items across a calendar
- Helps in **release planning** and **long-term strategy**



### **4. Custom Views**

- Save filtered views based on labels, assignees, or status
- Toggle between board and table layouts within the same project



# GitHub Projects - Configuration Options





Karan Gupta



# GitHub Projects Configuration Options

## 1. Fields Configuration

- **Custom Fields:** Create fields like priority, status, story points
- **Types:** Text, number, date, single/multi-select
- **Use Case:** Prioritize tasks, track estimates, or categorize by feature

## 2. Grouping & Sorting

- **Group By:** Status, assignee, label, or custom fields
- **Sort By:** Title, creation date, due date, or priority
- Helps create **focused views** for specific teams or workflows

A circular icon containing a stylized profile of a person's head, with a small globe icon at the top left.

Karan Gupta

### **3. Automation Rules**

- Auto-move items based on triggers (e.g., issue closed → "Done")
- Examples:
  - Assign labels or users automatically

### **4. Permissions & Visibility**

- **Access Control:** Public, private, or organization-restricted
- **Role-Based Permissions:** Admins, editors, and viewers

### **5. Templates**

- Start with a **blank project** or use pre-built templates like "Team Plan" or "Bug Triage"



# GitHub Projects vs Classic Projects



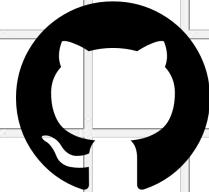
# GitHub Projects vs Classic Projects

Feature	GitHub Projects (New)	Projects Classic
User Interface	Modern, spreadsheet + kanban views	Basic kanban-style board
Custom Fields	<input checked="" type="checkbox"/> Supports multiple field types	<input type="checkbox"/> Not supported
Automation	<input checked="" type="checkbox"/> Advanced automation rules	Limited automation
Views	Table view, board view, saved views	Only board view
Filters & Sorting	Dynamic filtering and sorting	Very limited
Permissions	Role-based access (view, edit, admin)	Simple repository-level permissions
Templates & Scalability	Supports templates, scalable for orgs	Limited template options



# Labels vs Milestones

## Usage



# Labels & Milestones – GitHub Projects



## → Labels

Purpose: Categorize and filter issues/PRs

### Examples:

- bug, enhancement, documentation, priority: high

### Usage in Projects:

- Filter views based on labels (e.g., show only bug issues)
- Group work items by label in table or board view

bug	Something isn't working
documentation	Improvements or additions to documentation
duplicate	This issue or pull request already exists
enhancement	New feature or request
good first issue	Good for newcomers
help wanted	Extra attention is needed
invalid	This doesn't seem right
question	Further information is requested
wontfix	This will not be worked on



## Milestones



Karan Gupta



**Purpose:** Track progress toward a goal or release

**Includes:** Start/end dates, related issues/PRs

**Progress Tracking:** Shows percentage complete

### Usage in Projects:

- Align issues/PRs to a milestone (e.g., v1.0 Release)
- Prioritize and track sprint goals or version targets

The screenshot shows a project management interface with a grid background. At the top, there are tabs for 'Labels' and 'Milestones', with 'Milestones' being the active tab. A green button labeled 'New milestone' is visible. Below the tabs, a summary shows '2 Open' and '1 Closed'. A specific milestone card for 'beta release' is displayed, showing it has no due date and was last updated 3 days ago. The card includes a progress bar indicating '50% complete' with '1 open' and '1 closed' issues. Buttons for 'Edit', 'Close', and 'Delete' are at the bottom of the card.

Labels

Milestones

New milestone

2 Open ✓ 1 Closed

Sort ▾

beta release

No due date Last updated 3 days ago

50% complete 1 open 1 closed

Edit Close Delete



# GitHub Project Workflows



- Workflows are **customized automation rules** that help manage the lifecycle of issues and pull requests within GitHub Projects.
- Think of them as **if-this-then-that** logic for project boards.

## How to Set Up a Workflow:

1. Open your project
2. Click on “...” → “Workflows”
3. Choose a **trigger** (e.g., Issue created, PR closed)
4. Add **conditions** (e.g., has label **bug**)
5. Define **actions** (e.g., move to "To Do", assign user)



Karan Gupta



## Common Workflow Examples:

- On Issue creation → Add to “Backlog” column
- When PR is opened → Move to “In Review”
- When Issue is closed → Move to “Done”
- If label = **urgent** → Add to “High Priority”

## Benefits:

- Saves manual effort
- Ensures consistent task flow
- Improves tracking and response time



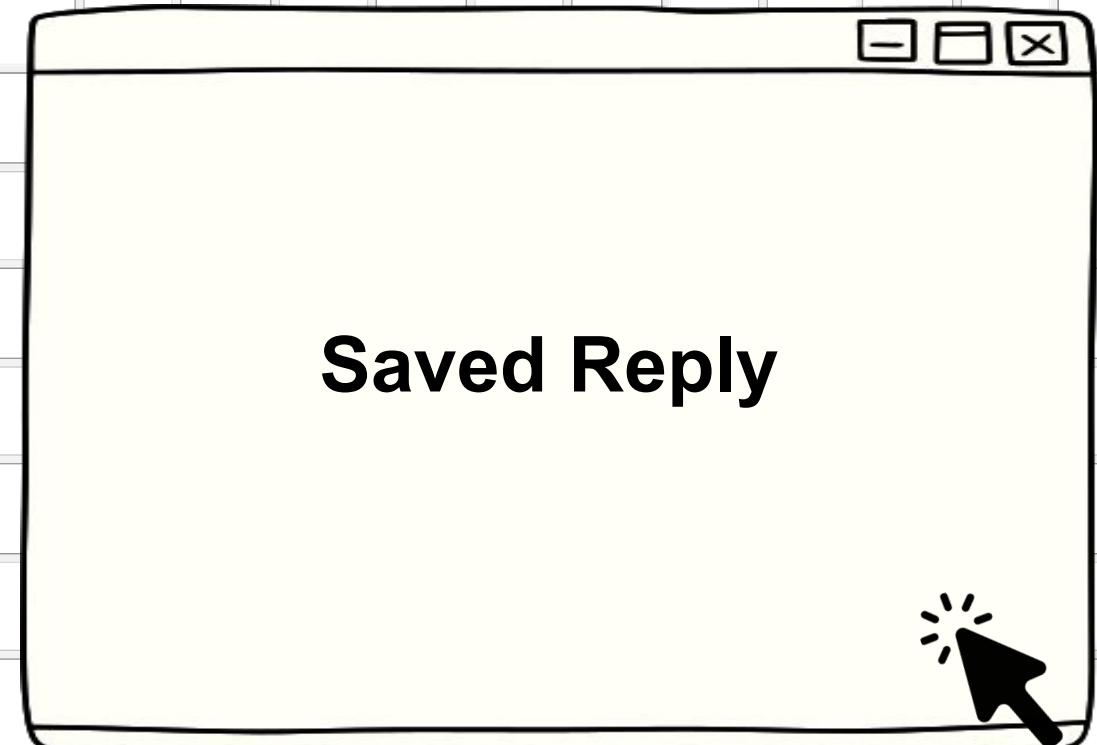
# GitHub Project Insights



- **Project Insights** provide **real-time analytics and reporting** within GitHub Projects to help teams monitor progress, identify bottlenecks, and make informed decisions.

## Key Insights:

- **Item Count by Status**
  - See how many tasks are in To Do, In Progress, or Done
- **Burndown Charts (for milestones)**
  - Track remaining work over time
- **Velocity Tracking**
  - Measure how much work is completed per iteration



Karan Gupta



# GitHub Project Saved Reply

A **Saved Reply** is a reusable, pre-written message that you can quickly insert into comments on:

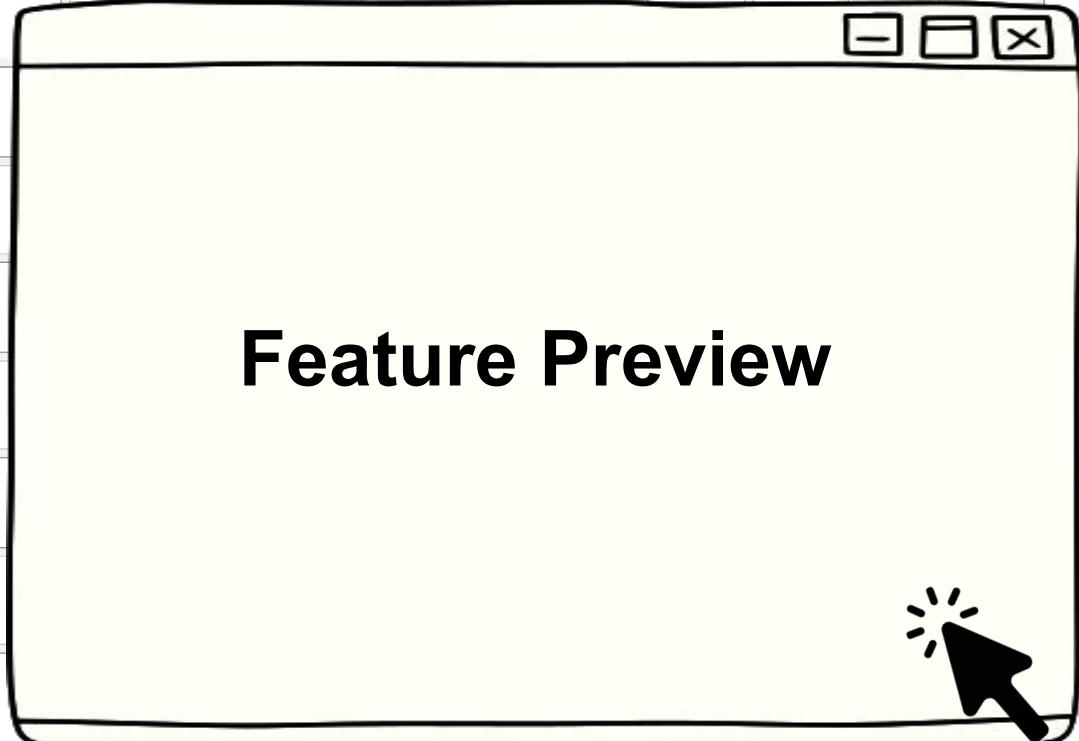
- Issues
- Pull Requests
- Discussions

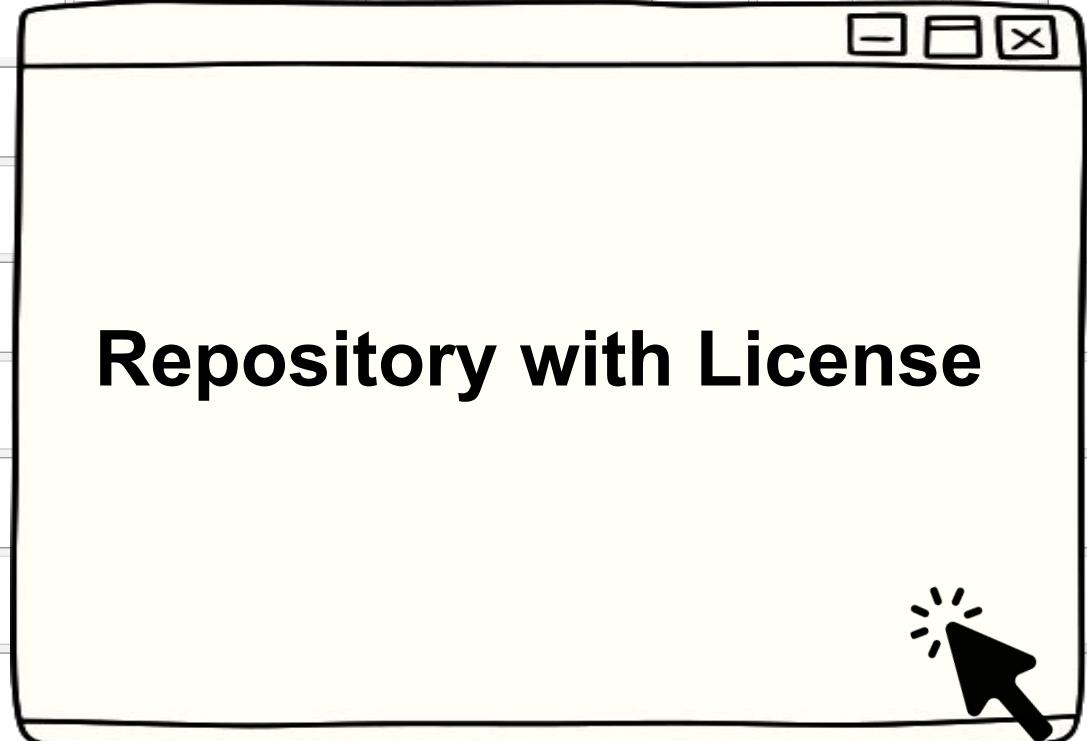
This helps streamline communication, especially for repetitive responses.

## Use Cases:

- Acknowledging new issues: *"Thanks for reporting! We'll look into this."*
- PR reviews: *"Thanks for your PR! We'll review and get back soon."*
- Redirects: *"Please follow the issue template or provide steps to reproduce."*









Karan Gupta

# GitHub Repository with License

A **license** in a GitHub repository defines:

- **How others can use**, modify, share, or distribute your code.
- **Your rights** as the original author.
- **User responsibilities** when using your work.

License	Permissions	Limitations
MIT	Free use, modify, distribute	Must include original license
Apache 2.0	Same as MIT + patent protection	Must state changes
Unlicense	Public domain	No warranty





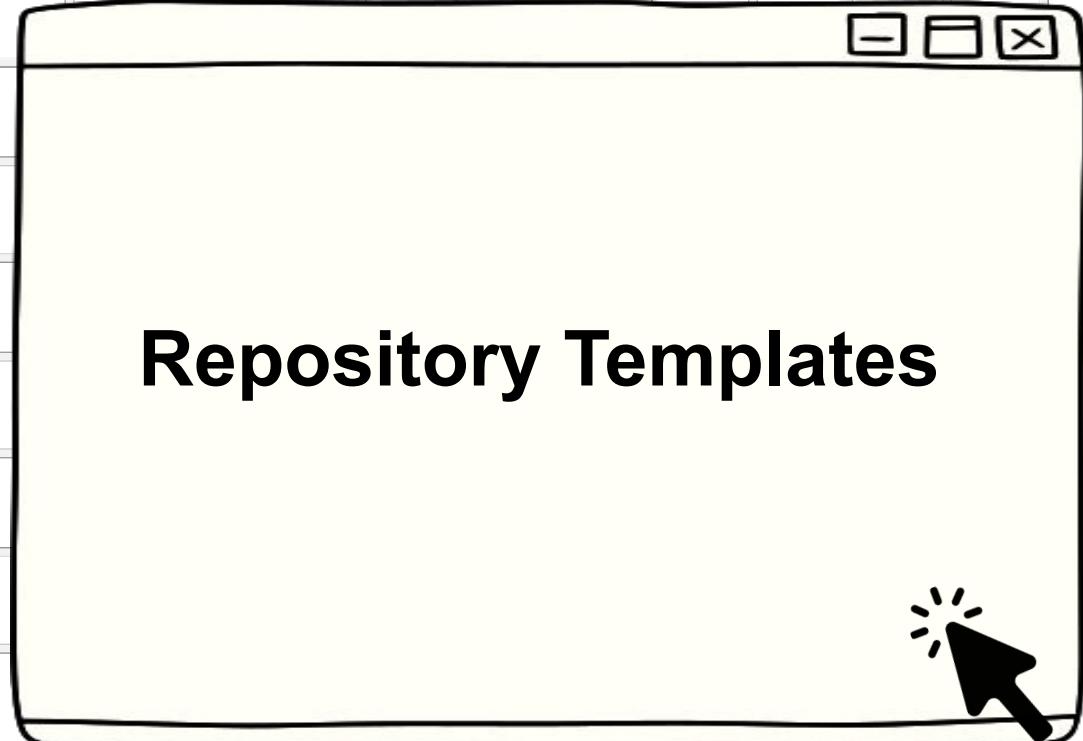
Karan Gupta



# GitHub Repository Insight

- GitHub Repository Insights provide **metrics and visualizations** that help understand a repository's **activity, health, and contribution patterns**.

Insight Area	Description
Pulse	Summary of recent activity: PRs, issues, commits, and contributors.
Contributors	Graph of contributors and their commit history over time.
Commits	Frequency of commits across days/weeks/months.
Code Frequency	Lines of code added vs. removed weekly.
Dependency Graph	Shows dependencies and dependents
Security Advisories	Shows security alerts, CVEs, and patch suggestions.



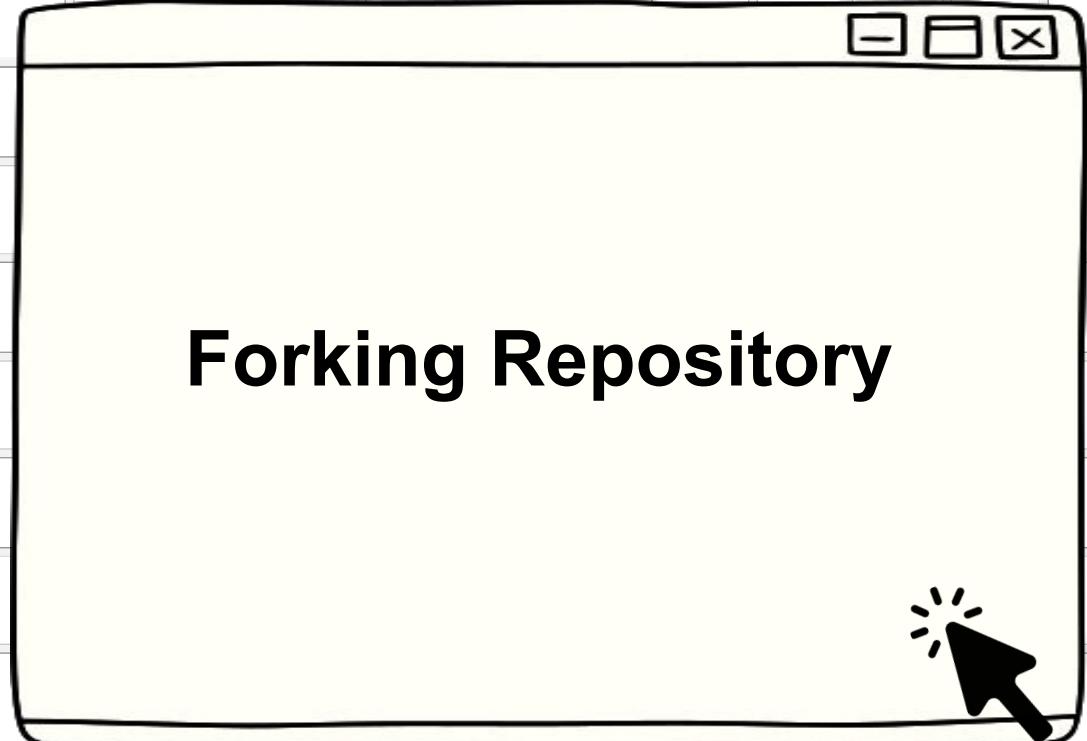
# GitHub Repository Templates



- A **Repository Template** is a special type of GitHub repository that lets you **create new repositories with the same structure, files, and settings**.
- Ideal for teams that want to **enforce standards** across multiple projects.

## Benefits of Using Repository Templates:

1. **Quick Project Setup**
  - Start a new repo with pre-configured code, folders, README, CI/CD, etc.
2. **Consistency Across Teams**
  - Maintain uniform structure across projects (e.g., dev guides, licenses)
3. **Great for Starter Projects**
  - Spin up similar services or student projects with ease



# GitHub Repository – Fork



Karan Gupta



- Forking creates a **personal copy** of someone else's GitHub repository under your own account.
- It allows you to **freely experiment, make changes**, and **submit improvements** without affecting the original project.

## Benefits of Using Repository Forks:

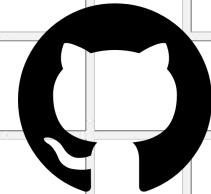
-  **Make Edits Without Risk**
  - Safely test new features or bug fixes
- **Contribute to Open Source**
  - Make a pull request from your fork to suggest changes
- **Collaborate with Teams**
  - Team members can fork, work in parallel, and submit PRs



# Starring & Watching



git



# GitHub Repository – Starring & Watching

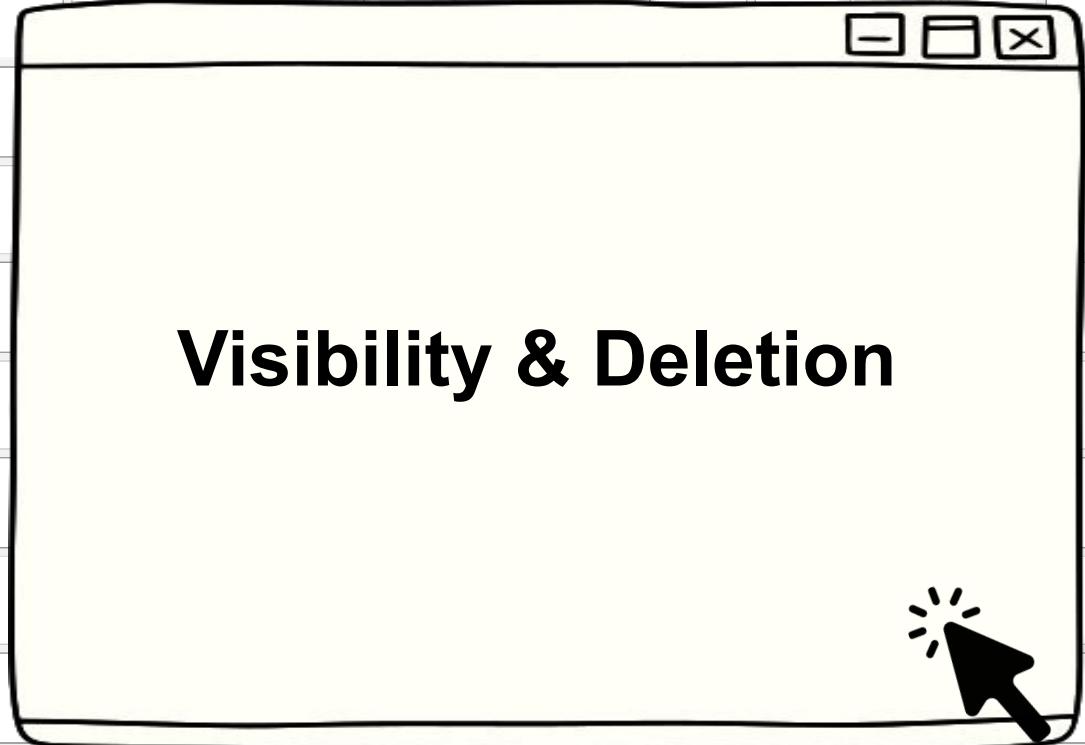


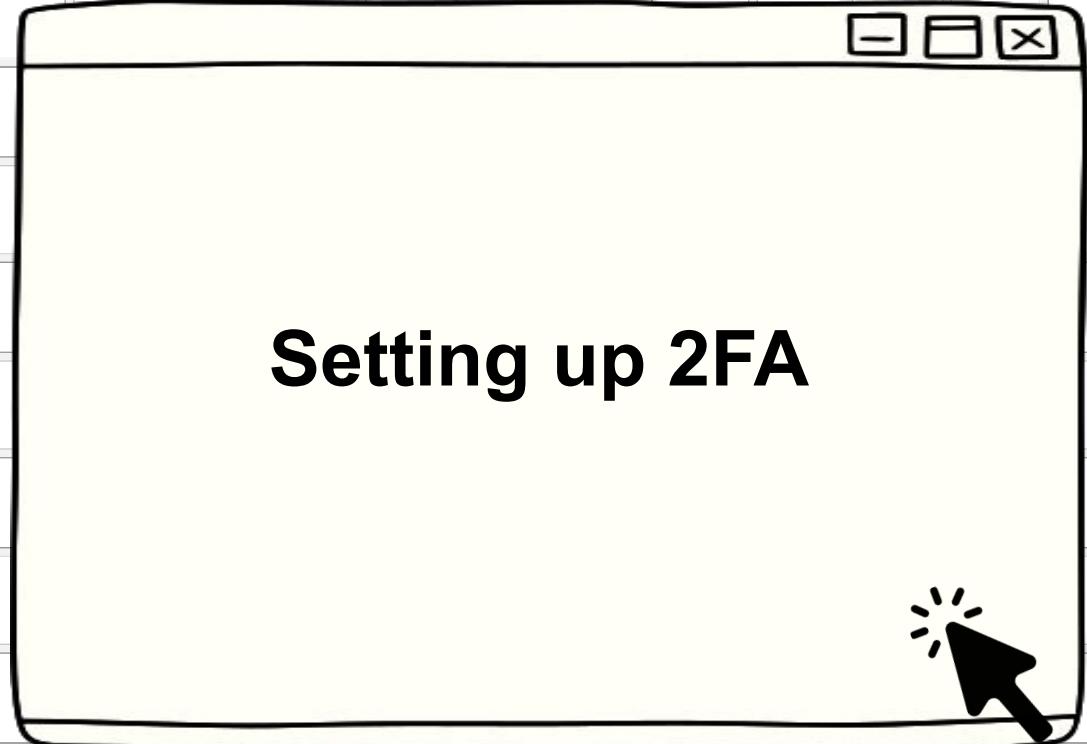
## Starring a Repository

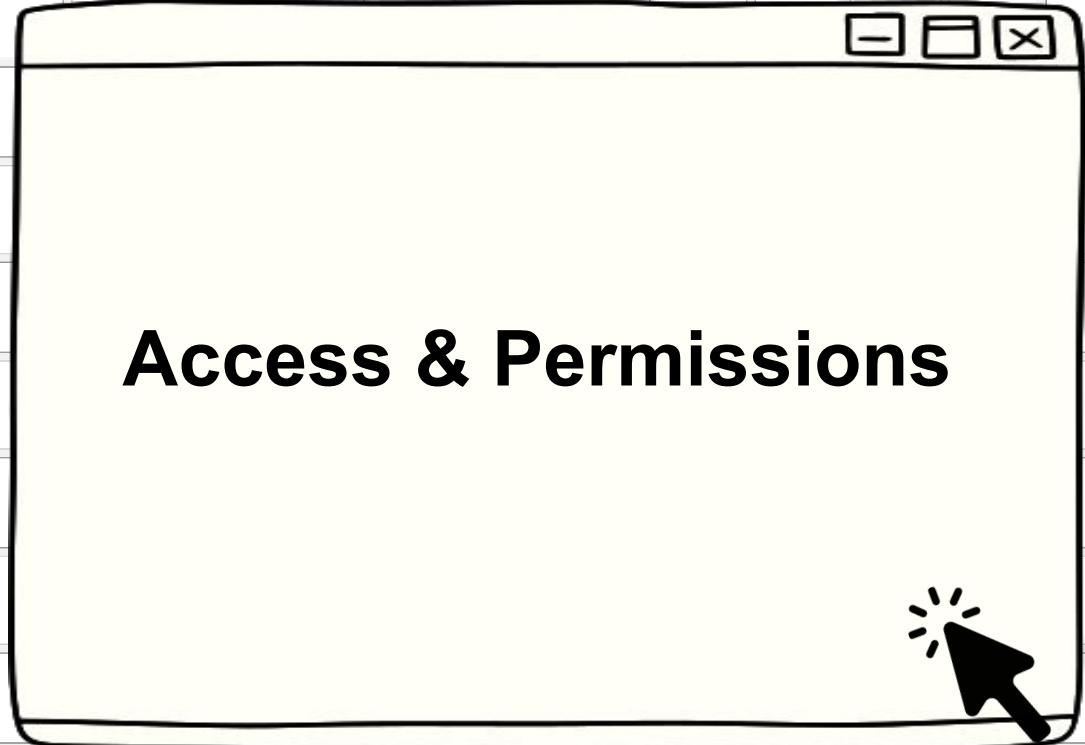
- **Purpose:** Show appreciation or bookmark a repo
- **Effect:** Adds the repo to your **Stars** list
- **Use Case:**
  - Track favorite open-source projects

## Watching a Repository

- **Purpose:** Receive **notifications** for activity
- **Options:**
  - **All Activity:** Issues, PRs, comments, releases
  - **Participating:** Only when you're mentioned or involved
  - **Ignore**







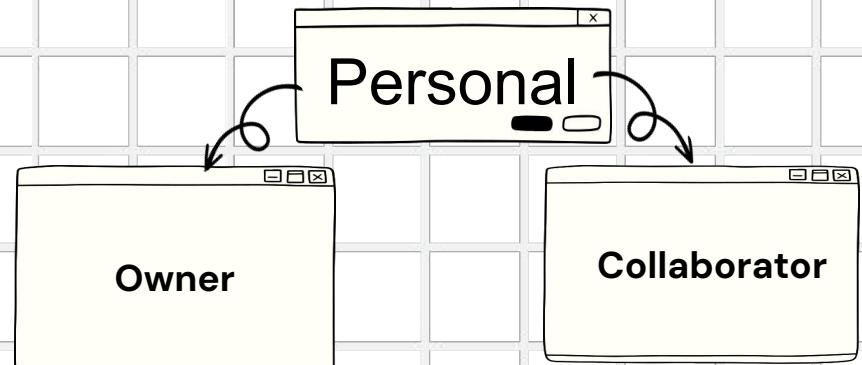
# Roles & Permissions



## Personal Accounts

### → Owner

- Full control over the repository
- Can:
  - Manage code, settings, access, secrets
  - Delete the repository



### → Collaborators

- Invited by the owner
- Collaborators on a personal repository can pull (read) the contents of the repository and push (write) changes to the repository.



Karan Gupta



## Organization Accounts

Role	Permission Scope
Read	View repo, clone, comment on issues/PRs
Triage	Manage issues and PRs (label, assign, close)
Write	Push commits, create branches, open/merge PRs
Maintain	Manage repo settings (without admin rights)
Admin	Full control: repo settings, teams, secrets, branch protections, etc.



# Enterprise Managed Users



Karan Gupta



- Enterprise Managed Users (EMUs) are GitHub accounts managed via your company's identity provider (e.g., Azure AD).
- EMUs are **created, provisioned, and controlled** using **SCIM & SAML-based SSO**.

Feature	Details
Identity Management	EMUs can't sign up manually — they're auto-provisioned
Centralized Control	Admins manage usernames, team memberships, and access
Corporate Identity Binding	GitHub accounts tied to company identities (e.g., <a href="mailto:john.doe@company.com">john.doe@company.com</a> )
Data Ownership	All content created by EMUs is owned by the <b>enterprise</b>
Lifecycle Automation	Auto-disable or remove users based on HR changes





# Personal Account Security

## Features



# Personal Account

Feature	Available in Personal Account
SSH/PAT Authentication	✓
Code & Secret Scanning	✓
Dependabot	✓
Branch Protection	✓
Workflow Secrets	✓
Security Policies	✓



# Personal Account Security

## Features HandsOn





# Organization Account Security Features

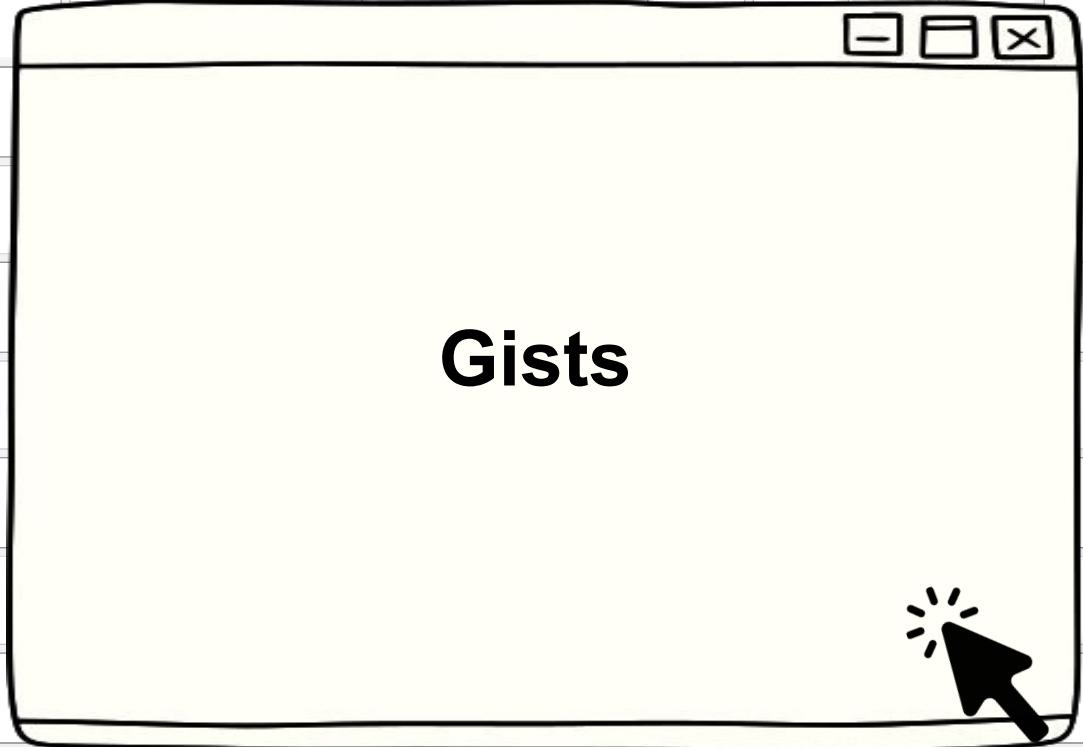


	<b>Feature</b>		<b>Available in Organization Account</b>
	2FA Enforcement		✓
	SSO via SAML		✓
	Audit Logs		✓
	Secret Scanning (private repos)		✓
	Code Scanning Automation		✓
	IP Allow List		✓
	Centralized Security Dashboard		✓
	Role/Team Management		✓
	Workflow & Secret Protection		✓
	Repository Rulesets & Controls		✓



# Organization Account Security HandsOn





# Gists



- A **Gist** is a simple way to share code snippets, notes, or config files.
- Hosted by GitHub and accessible at: <https://gist.github.com>

Type	Visibility	Description
Public	Visible to everyone	Indexed by search engines
Secret	Only with the link	Hidden from public search

## Use Cases:

- Share config files, scripts, or commands
- Document small code examples
- Use for online portfolios or resumes (with Markdown)

## Managing Gists:

- Edit, fork, delete, or comment
- Version history is auto-tracked
- Clone like a Git repository using:

```
git clone https://gist.github.com/yourgistid.git
```

## Features:

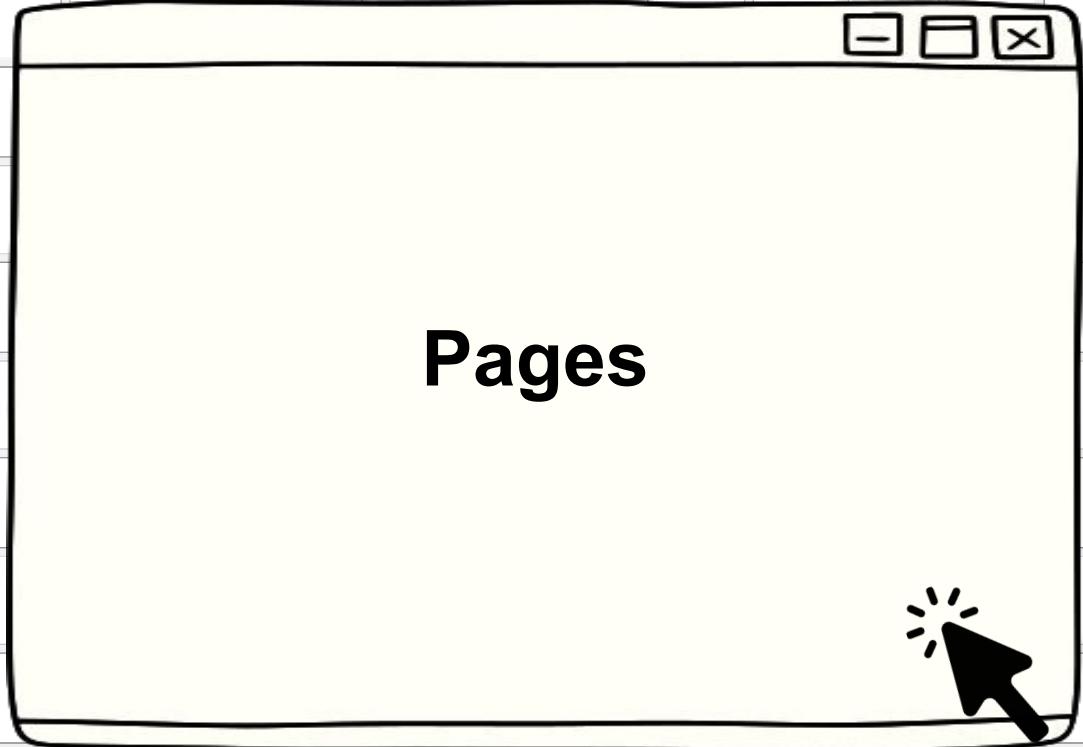
- Supports syntax highlighting
- Markdown rendering
- Shareable URLs
- Embed in websites/blogs



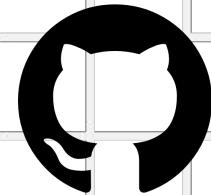
Karan Gupta





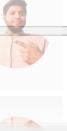


git





Karan Gupta



# Pages

- GitHub Pages is a **free hosting service** from GitHub that allows you to **publish static websites** directly from a GitHub repository.

## Use Cases:

- Project documentation
- Personal/portfolio sites
- Blogs and resumes
- Open-source project landing pages



Karan Gupta



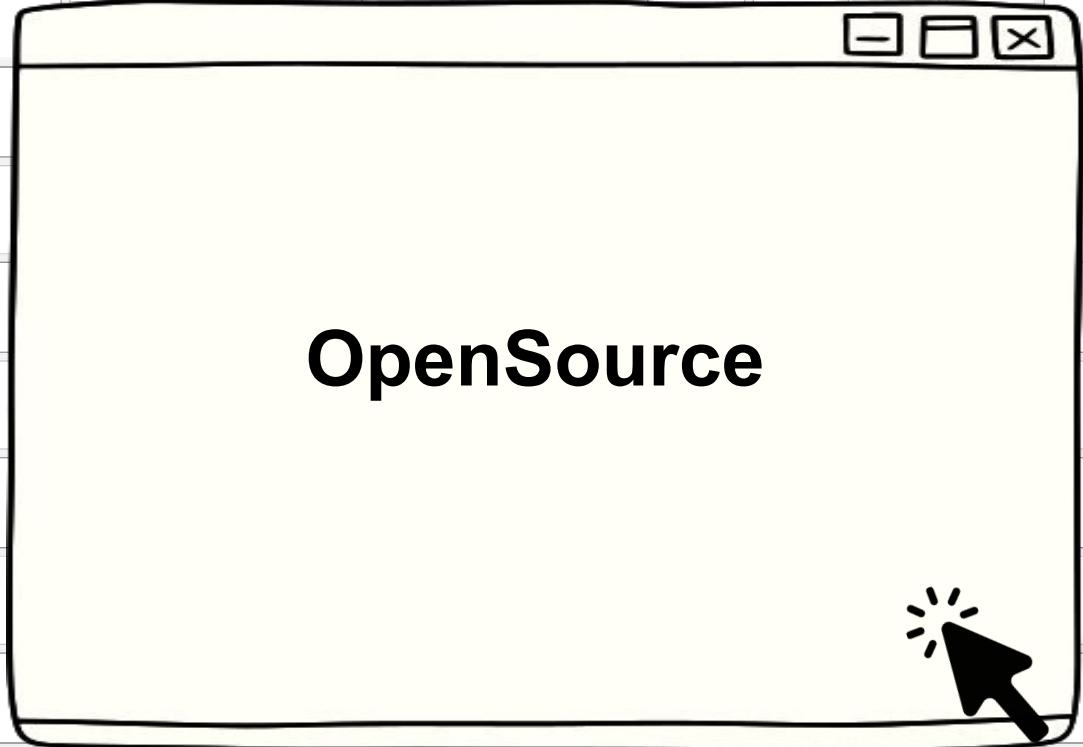
## **Features:**

- Free custom website for projects or profiles
- Supports HTML, CSS, JS, Jekyll, Markdown
- Integrates with GitHub Actions for CI/CD
- Uses GitHub repository as the source

## **URL:**

[https://<username>.github.io/<repo>/](https://<username>.github.io/<repo>)





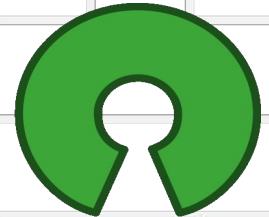


# OpenSource

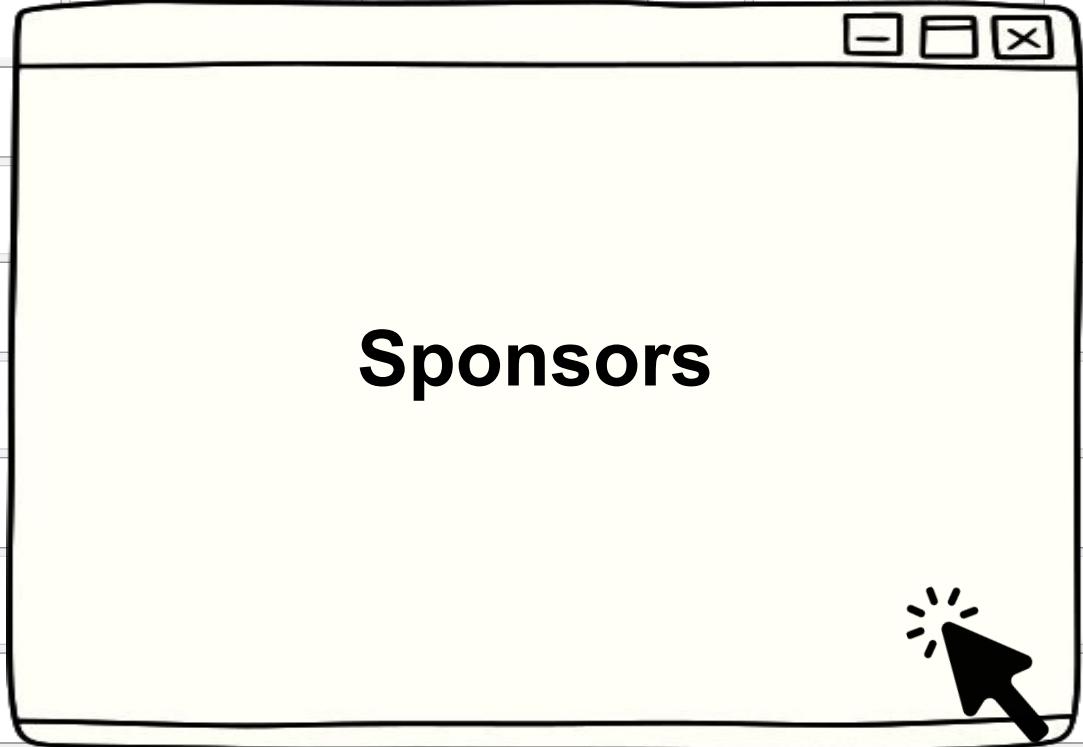
- Open Source means the source code is **freely available to view, modify, and distribute** under a license.

## GitHub: The Home of Open Source

- GitHub is the **largest platform for open-source collaboration**
- Hosts millions of public repositories across all programming languages
- Supports collaboration through **issues, pull requests, forks, and discussions**



**open source  
initiative®**





# Sponsors

- **GitHub Sponsors** is a feature that enables developers and organizations to **financially support open-source contributors** directly on GitHub.

## Sponsor Benefits:

- Sustain open-source projects
- Encourage community contributions
- Highlight sponsors in READMEs or websites
- Create a consistent income stream for maintainers

Karan Gupta

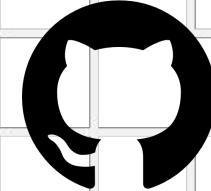


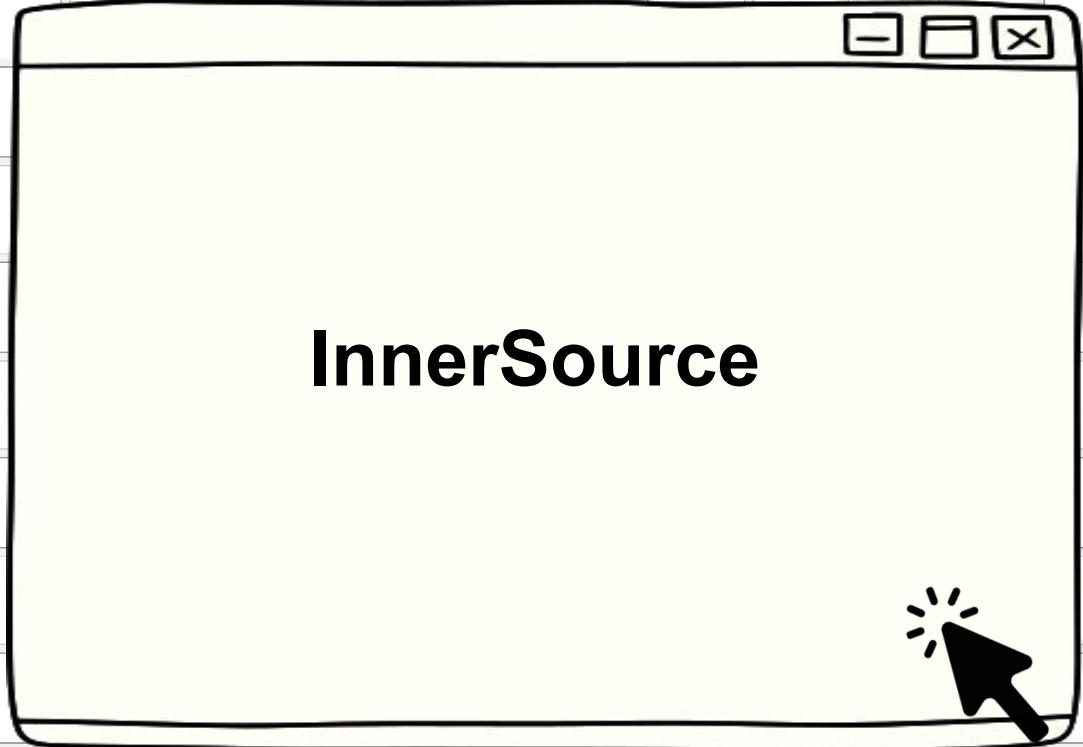
Feature	Description
<b>Support Developers</b>	Anyone can sponsor maintainers of public repos
<b>Recurring Payments</b>	Monthly sponsorships with customizable tiers
<b>Sponsorship Tiers</b>	Maintainers can define tiers with benefits (e.g., early access)
<b>Secure &amp; Transparent</b>	GitHub handles transactions, fees, and privacy settings
<b>No Fees</b>	GitHub takes <b>zero fees</b> for individual contributors (currently)



# Following People & Organisations



 git





# InnerSource

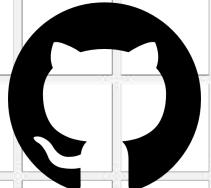
- **InnerSource** is the practice of applying **open source software development principles**—such as collaboration, transparency, and code sharing—**within an organization**.
- Think of it as running your internal projects like open source, but behind your company firewall.

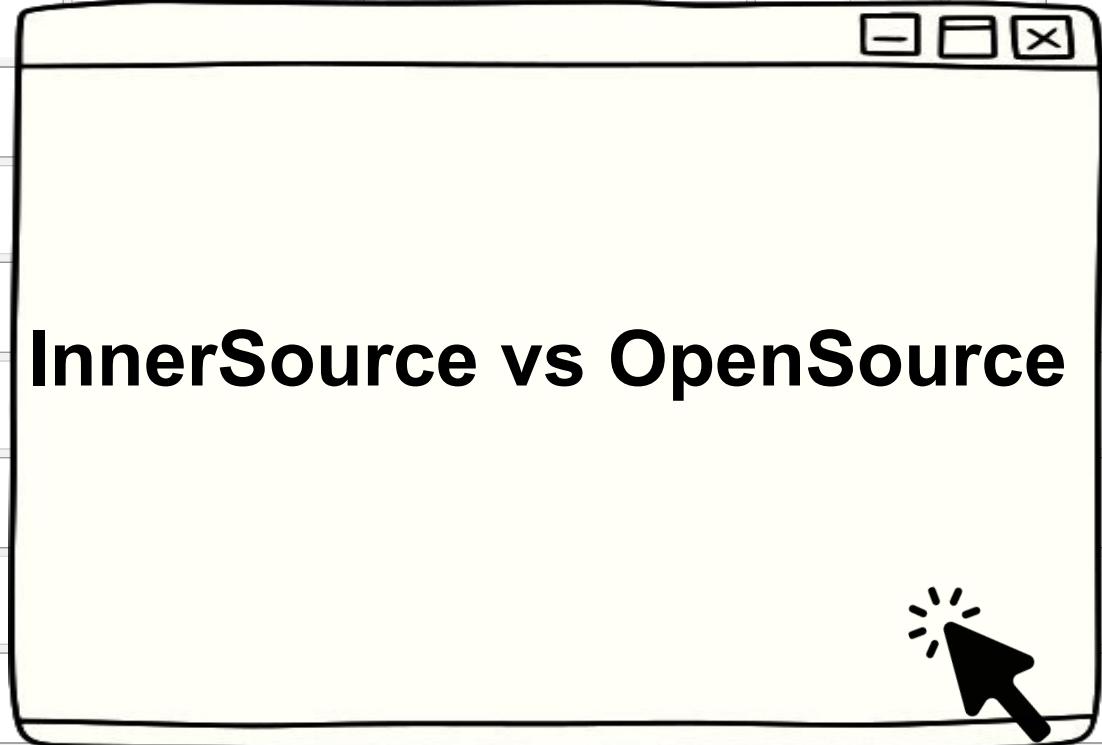
Principle	Description
<b>Open Collaboration</b>	Any team can contribute to any project across the org
<b>Transparency</b>	Code, issues, and documentation are visible internally
<b>Standard Workflows</b>	Uses GitHub-style tools like pull requests, reviews, and CI/CD
<b>Shared Ownership</b>	No "silos"—teams work together to build, fix, and improve software



## Features:

- **Accelerates innovation** by sharing solutions across teams
- **Leverages collective expertise** within the organization
- **Improves code quality** through peer reviews and standards
- **Breaks down silos**, encourages cross-team collaboration
- **Reduces duplication** of effort across departments





<b>Feature</b>	<b>InnerSource</b>	<b>Open Source</b>
<b>Visibility</b>	Code is visible <i>internally</i> within an org	Code is visible to <i>anyone</i> on the internet
<b>Contributors</b>	Internal employees only	Anyone in the global developer community
<b>Governance</b>	Controlled by internal teams or company rules	Often community-driven or foundation-led
<b>Access Control</b>	Uses company's access policies and tools	Public by default, but can be licensed for control
<b>Purpose</b>	Improve internal collaboration and reuse	Share solutions, collaborate globally
<b>Examples</b>	Internal tools, services, shared libraries	Linux, React, Kubernetes, etc.
<b>Tools</b>	GitHub Enterprise, GitLab (private), Bitbucket	GitHub (public), GitLab (public), SourceForge, etc.





Karan Gupta



# GitHub Wiki

- GitHub Wiki is a **built-in documentation tool** that allows you to create **well-structured, collaborative documentation** directly within your GitHub repository.

## Key Features:

- Supports **Markdown** and **rich formatting**
- Acts like a **mini website** for your project
- Create **multiple pages** and organize them with links
- Automatically version-controlled with Git

## Use Cases:

- Project setup guides
- Developer documentation
- API references
- Contribution guidelines
- Team notes

CourseDIY / fast-forward

Type to search

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

**First Welcome Page**

CourseDiy edited this page 8 minutes ago · 1 revision

Welcome to the fast-forward wiki!

**This is hello world program.**



Pages 2

Find a page...

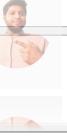
Home

First Welcome Page  
This is hello world program.

+ Add a custom sidebar

Clone this wiki locally

<https://github.com/CourseDIY/fast-forward>



Karan Gupta







Karan Gupta



# Discussions

- GitHub Discussions is a collaborative space **within a repository** for open-ended conversations, community Q&A, and planning.

## Key Features:

- Open, threaded conversations
- Markdown support with rich formatting
- Can be categorized (e.g., Q&A, Ideas, Announcements)
- Answers can be marked as **accepted**
- Integrates with Issues and Pull Requests

## Use Case:

- Encourage community engagement
- Avoid cluttering Issues with non-actionable topics
- Share ideas, feedback, and plans publicly



Karan Gupta



Feature	Benefit
Ideas	Gather feedback or feature requests
Q&A	Community-supported support channel
Announcements	Share updates or roadmaps
Cross-linking	Link to Issues/PRs for context



# Discussions HandsOn





# Markdown



Karan Gupta



Markdown is a **lightweight markup language** used to format plain text into structured documents like:

- README files
- GitHub comments, Issue, Pull Request
- Wiki pages or Documentation

## Benefits:

- Easy to write and read
- Supported natively on GitHub
- Works well for technical and project documentation
- Converts automatically to HTML

Task	Markdown Syntax	Output
Heading	#, ##, ###	# Heading
Bold	**bold**	<b>bold</b>
Italic	*italic*	<i>italic</i>
Link	[text](url)	<a href="#">GitHub</a>
List	- Item or 1. Item	Bullet/Numbered list
Code (inline)	`code`	code
Code Block	<code>code</code>	Code block
Image	![alt](url)	Image



# Enterprise - Deployment Options



# Enterprise Deployment Options



## 1. GitHub Enterprise Cloud

- **Hosted by GitHub** (SaaS solution)
- Integrates directly with [github.com](https://github.com)
- **No infrastructure management** needed
- Supports **Enterprise Managed Users (EMUs)** for identity control
- Seamless access to **GitHub Actions, Pages, Copilot, Codespaces**, etc.
- Ideal for orgs looking for **scalability and simplicity**

 **Best for:** Cloud-first companies, startups, remote teams

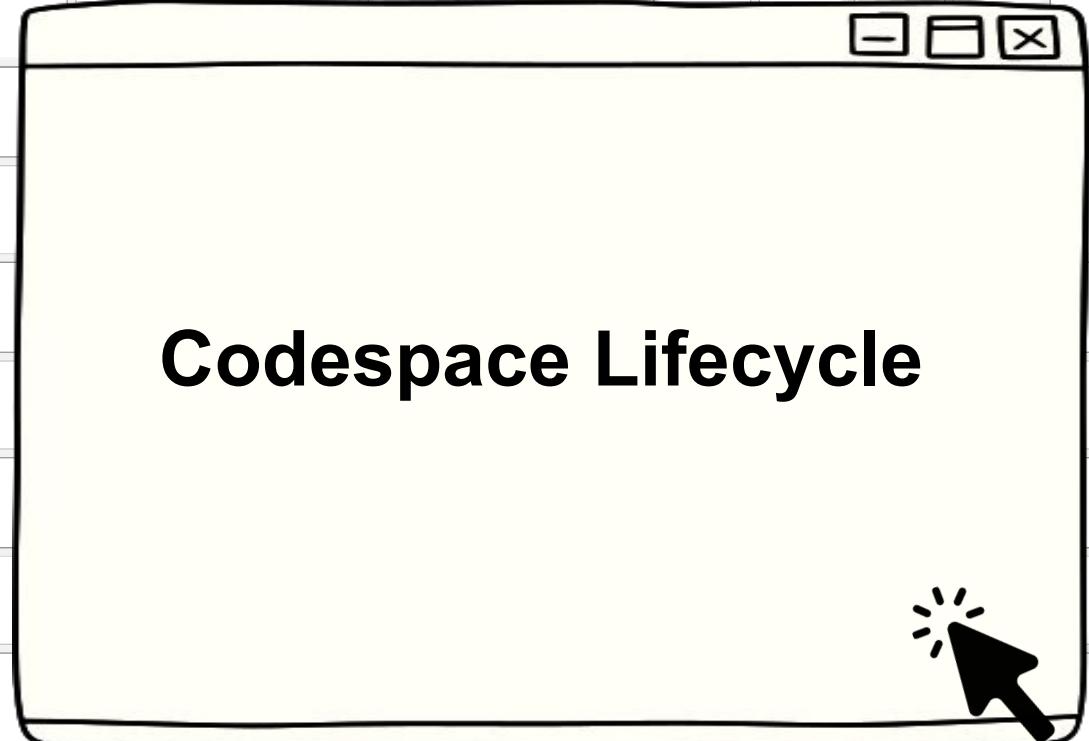
## 2. GitHub Enterprise Server

- **Self-hosted** GitHub instance on your own infrastructure or cloud provider (e.g., AWS, Azure)
- Runs on **virtual machines** using an **OVA, AMI, or Azure image**
- Full control over **data, storage, network, and backups**
- Requires internal DevOps to manage scaling, updates, and monitoring
- Supports **air-gapped environments** and custom integrations



**Best for:** Regulated industries, on-prem environments, data-sensitive orgs

Feature	Enterprise Cloud	Enterprise Server
Hosted by	GitHub	Your Org / Cloud Provider
Infrastructure	Managed by GitHub	Self-managed
Access to GitHub Features	Full	Most (some features limited)
EMUs	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No (SAML-based SSO instead)
Offline Support	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes
Ideal For	Cloud-first orgs	Regulated, on-prem orgs



# Codespace Lifecycle

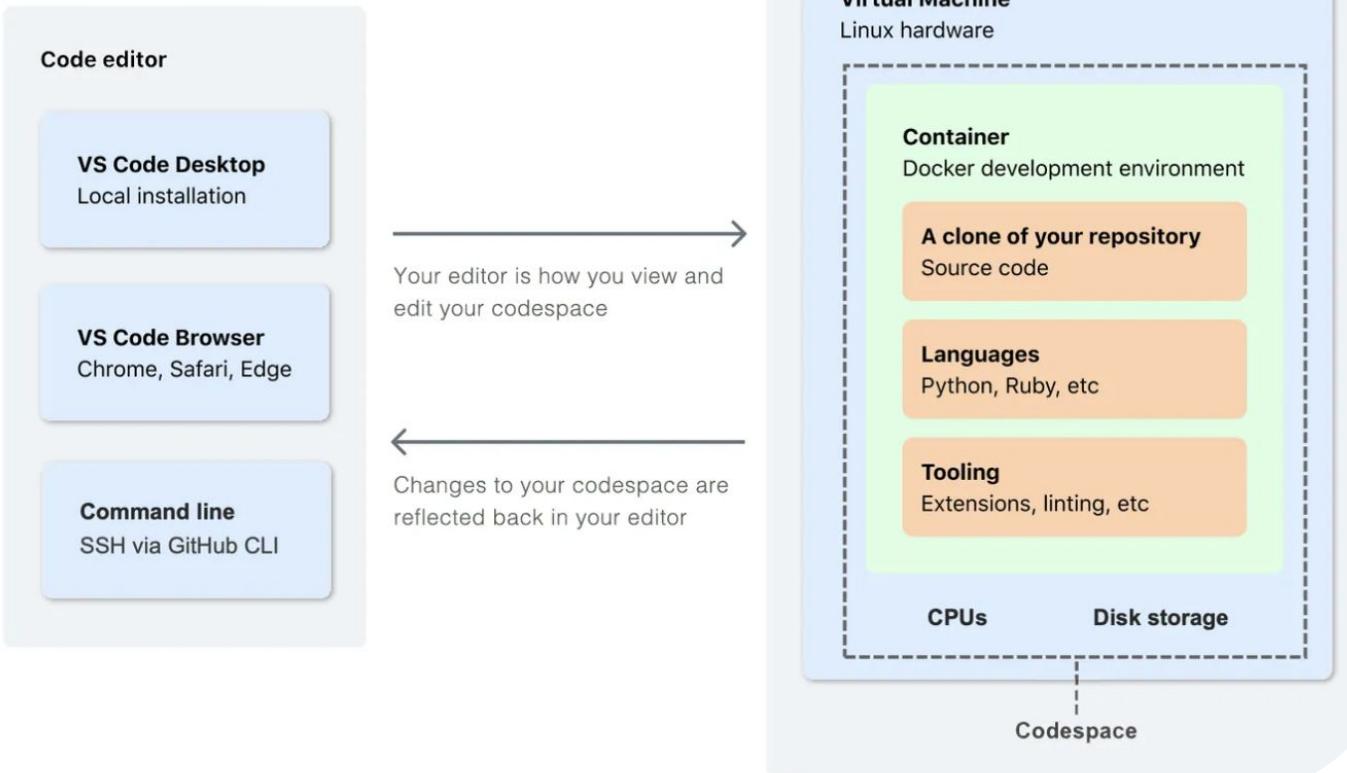


## 1 Creation

- Uses a repository's **dev container configuration** (`.devcontainer/` folder).
- Installs dependencies, extensions, and setups the runtime environment.
- Starts a VS Code-based environment in the cloud.

## 2 Running

- Developer interacts with the environment:
  - Write code
  - Run & debug applications



### **3**Suspend (Auto-Pause)

- If inactive for a set time (default: 30 minutes), the Codespace is **paused** to save compute.
- State is saved, so work is not lost.

### **4**Resume

- When accessed again, the environment is **resumed** in seconds.
- All previously opened files and context are restored.

### **5**Stop (Manual)

- You can **manually stop** a Codespace from the dashboard.
- Resources are freed, but the workspace and changes are saved.

### **6**Delete

- Removes: All files and environment



# Organisation Permission Level



# Organisation Permission Level

<b>Permission Level</b>	<b>Capabilities</b>
<b>Admin</b>	Full control — manage settings, collaborators, branches, webhooks, etc.
<b>Maintain</b>	Limited admin powers — can manage issues, PRs, releases, not settings.
<b>Write</b>	Can push commits, manage branches, create PRs, close issues.
<b>Triage</b>	Manage issues/PRs, apply labels, assign people — no code write access.
<b>Read</b>	View code, clone repo, open issues and PRs — no write access.



# Organisation Role Level



Role	Key Access
Owner	Full admin, billing, repos, members
Member	Repo access based on permissions
Moderator	Community moderation (issues, discussions, abuse)
Billing Manager	Billing only
Security Manager	Security features and settings
App Manager	GitHub Apps approval and configuration
Outside Collaborator	Limited repo access, no org-level visibility



Karan Gupta

## 1. Organization Owners

- Have full administrative access to the organization.
- Can manage members, repositories, billing, security settings, and integrations.

## 2. Organization Members

- Default role for internal users.
- Can access repositories and collaborate based on their assigned permissions.

## 3. Organization Moderators

- Help manage community interactions like discussions and issues.
- Can block/report users and enforce moderation policies (available in public repos/orgs).

## 4. Billing Managers

- Can view and manage billing information only.
- Have no access to code, repos, or organizational settings.



## 5. Security Managers

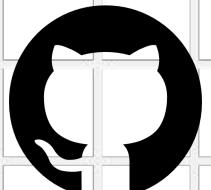
- Focus on repository and organization-wide security settings.
- Can manage Dependabot alerts, secret scanning, and security policies.

## 6. GitHub App Managers

- Can install, approve, and configure GitHub Apps within the organization.
- Manage app permissions and repo access for integrations and automation.

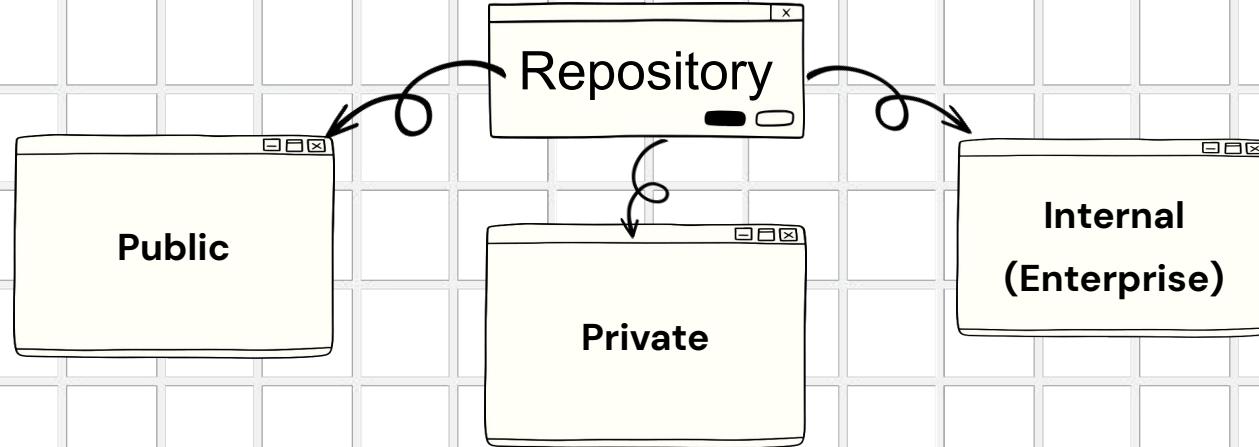
## 7. Outside Collaborators

- Users outside the organization with limited access to specific repositories.
- Cannot see or manage organization settings or internal resources.



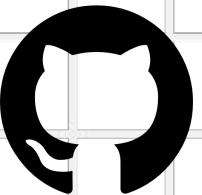


# Repository Visibility



## 1. Public Repository

- Visible to **everyone** on the internet.
- Ideal for **open-source projects**.
- Anyone can **clone, fork, and view** the code.





Karan Gupta



## 2. Private Repository

- Visible only to **invited collaborators or org members**.
- Perfect for **proprietary or internal projects**.
- Full control over **who can read/write/administer**.

## 3. Internal Repository (For GitHub Enterprise)

- Visible to **all members** of the organization.
- Not accessible to the public or outside collaborators.
- Useful for **intra-org collaboration** without full public exposure.



# Branch Protection Rules



Branch protection rules help enforce workflows and prevent unwanted changes to critical branches in your GitHub repository.

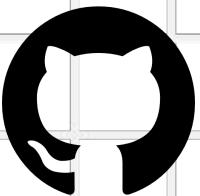
## → Common Branch Protection Settings

### 1. Require Pull Request Reviews

- Prevent direct commits to the branch by Enforcing code review before merging.
- Optionally, require **approved reviews** (e.g., at least 1 or 2 reviewers).

### 2. Require Status Checks to Pass

- Ensures CI/CD pipelines pass (e.g., GitHub Actions, Jenkins).
- Prevents merging broken code.



### **3. Require Signed Commits**

- Enforces GPG-signed commits for security/audit trail.

### **4. Restrict Who Can Push**

- Allow only specific users/teams/bots to push to the branch.

### **5. Require Linear History**

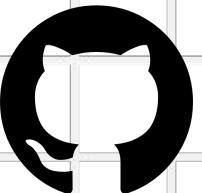
- Avoid merge commits; enforces rebase or squash.

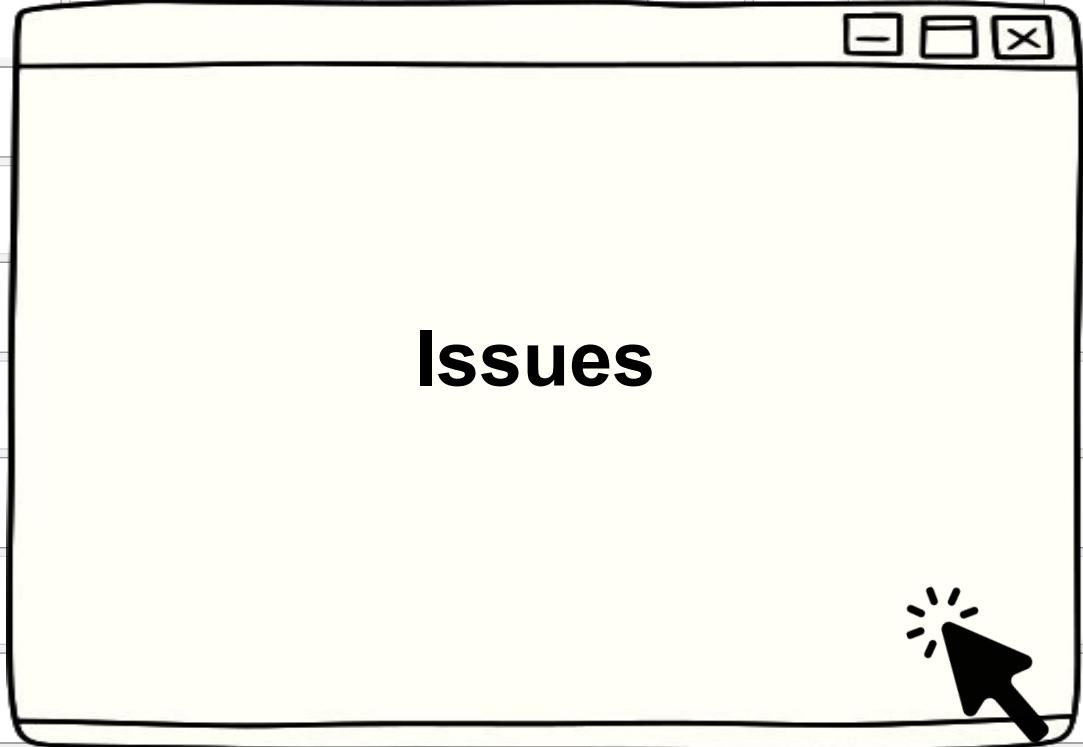
### **6. Require Conversation Resolution**

- All review comments must be resolved before merging.

### **7. Lock Branch**

- Disallows all pushes, even from admins.





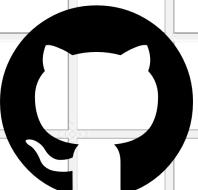
# GitHub Issues



Issues in GitHub are used to **track tasks, bugs, feature requests, and discussions** within a repository.

## 🎯 Key Features:

- Title & description (Markdown supported)
- Labels for categorization
- Assignees for ownership
- Milestones for timeline grouping
- Linked Pull Requests for context

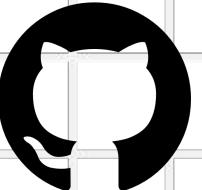


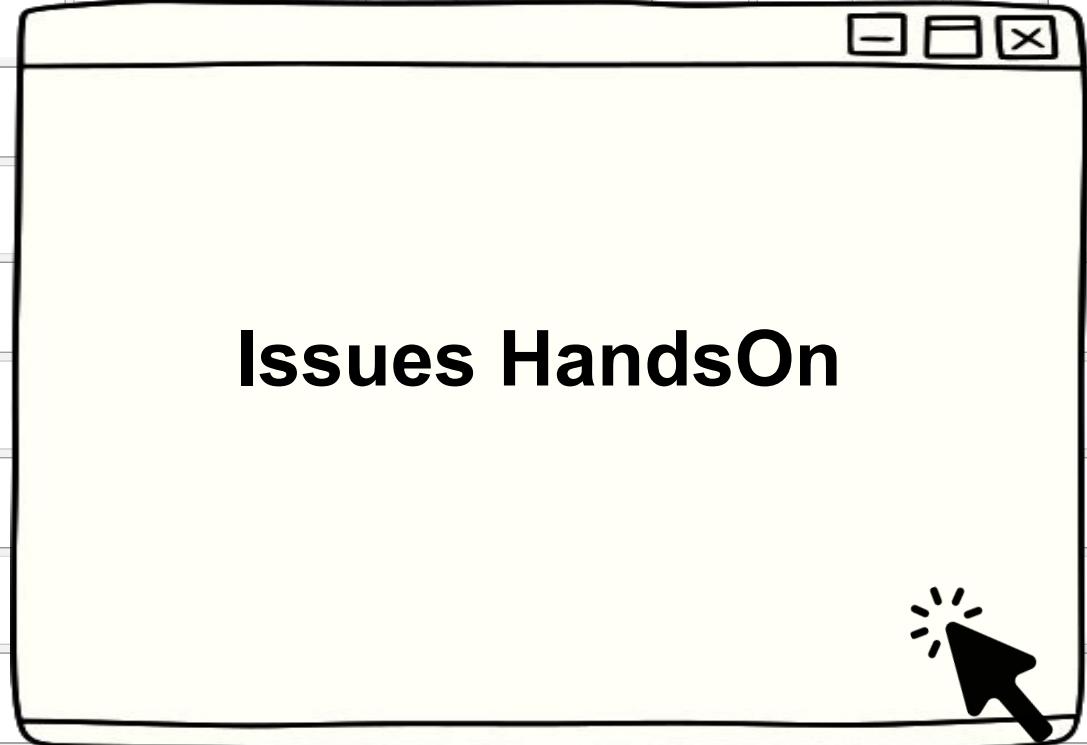
## **Common Use Cases:**

- Report bugs
- Plan new features
- Track development tasks
- Facilitate project discussions

## **Basic Issue Workflow:**

1. Create Issue
2. Assign labels, assignees, and milestone
3. Discuss and collaborate via comments
4. Link PRs and close when resolved







Karan Gupta

# Issues - Creating Branch HandsOn





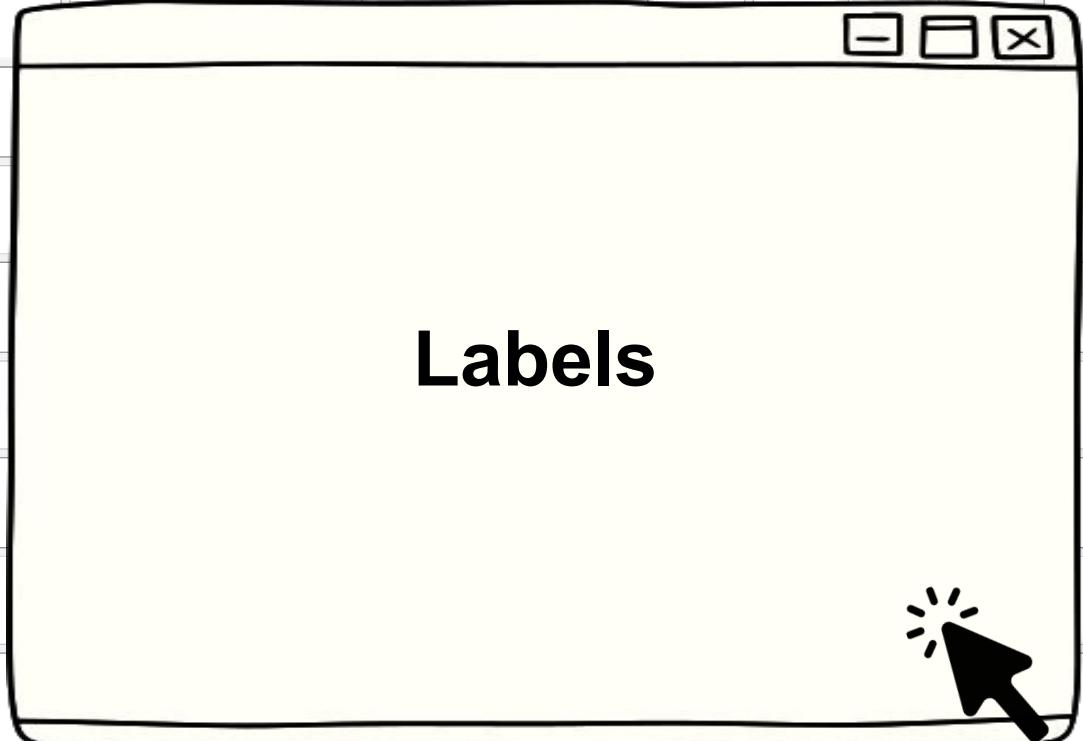
# Linking Issues to PR



- **Automatic Linking (Recommended)**
  - Use **keywords** in your **PR description** to **automatically close** the issue when the PR is merged.
    - [Fixes #123](#)
    - [Closes #456](#)
    - [Resolves #789](#)
- **Manual Linking (Non-Closing)**
  - If you just want to reference the issue without closing it:
    - [Related to #101](#)
    - [See also: #202](#)



Karan Gupta



# GitHub Labels

- **Labels** in GitHub help categorize and organize **issues** and **pull requests**.
- Think of them like **tags** that describe the type, priority, or status of an item.

## Purpose:

- Easily **filter and find** related issues/PRs.
- Help teams prioritize and track work.
- Improve project visibility and communication.

bug

documentation

duplicate

enhancement

good first issue

help wanted

invalid

question

wontfix



Karan Gupta



## Common Examples:

- **bug** – something is broken
- **enhancement** – a feature request
- **question** – someone needs help
- **help wanted** – contribution welcome

## Managing Labels:

- Owners/admins can **create**, **edit**, or **delete** labels.
- Each label can have:
  - A **name**
  - A **description**
  - A **color** (for visual grouping)





# GitHub Milestone

- A **milestone** in GitHub is a way to group related **issues** and **pull requests** that contribute to a larger goal — like a release or a sprint.

## **Purpose:**

- Helps track progress toward a big deliverable.
- Organizes work into phases or deadlines.
- Shows what's complete and what's pending.

## **Example:**

- Milestone: **v1.0 Release**
  - Issues: Fix login bug, Add user profile



Karan Gupta

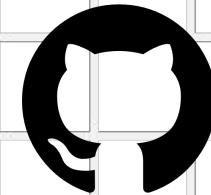


## **How to Use:**

- Create milestones via the “**Milestones**” tab in the **Issues** section.
- Assign issues or PRs to a milestone during creation or edit.

## **Milestone Details Include:**

- Title (e.g., **Sprint 5**)
- Description (goals or overview)
- Due date (optional)
- Progress bar (auto-calculated by closed vs. open issues)

 git

# GitHub Saved Replies

- **Saved Replies** are pre-written messages you can reuse when responding to **issues** and **pull requests**.

## Purpose:

- Save time by avoiding repetitive typing.
- Ensure consistent and professional communication.
- Improve team productivity and response quality.

## Example:

- Thanking contributors
- Explaining issue closing reasons



Karan Gupta



## How to Create:

1. Go to **Settings > Saved replies** (under your GitHub profile).
2. Click "**New saved reply**".
3. Add a **title** and **message content**, then save.

## How to Use:

- When replying to an issue or PR comment:
  - Click the **💬 icon (comment box)**.
  - Choose a saved reply from the dropdown.



# Types of Pull Request Status





Karan Gupta



# GitHub Pull Request Status:

- PR Status in GitHub tells you **if a pull request is ready to be merged** based on checks, reviews, and merge conditions.

## **Why It Matters:**

- Helps maintain code quality, enforce workflows, and prevent broken changes from being merged.

## **Where It's Shown:**

- Directly on the **PR page**, under the “Checks” and “Merge” sections.



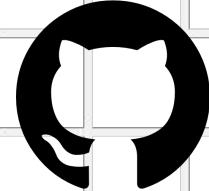
Karan Gupta



Status Type	Meaning
<b>Pass/Success</b>	All required checks (CI, tests, linting, etc.) passed.
<b>Fail/Error</b>	One or more checks failed – action is needed.
<b>Pending</b>	Some checks are still running – wait for completion.
<b>Blocked</b>	Cannot merge due to issues like review required or conflicts.
<b>Draft</b>	The PR is a draft and not ready for review/merge.
<b>Review Required</b>	At least one required code review is pending.



# Discussion vs PR vs Issues

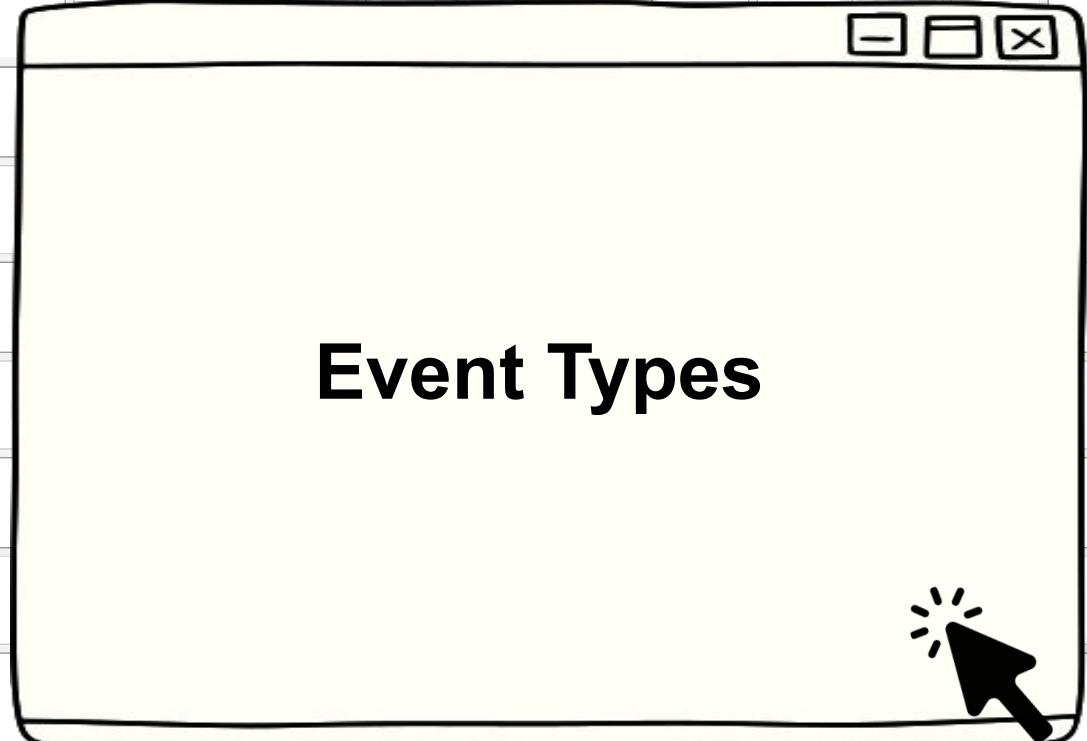
The word "git" in a large, bold, black sans-serif font. To the left of the text is a red diamond-shaped icon containing a white branching line symbol, representing Git.

<b>Feature</b>	<b>Discussion</b>	<b>Pull Request (PR)</b>	<b>Issue</b>
<b>Purpose</b>	Open-ended conversations, feedback, Q&A, brainstorming	Propose code changes, review and merge code	Track bugs, tasks, feature requests, improvements
<b>Common Use Cases</b>	Community interaction, asking for advice or ideas	Code review, feature or bug fixes	Reporting bugs, feature requests, general tasks
<b>Related to Code</b>	Not directly related to code changes	Directly related to code changes (merging branches)	Can be linked to PRs, commits, or specific code changes
<b>Interaction Type</b>	Threaded discussion, open collaboration	Review comments, approvals, merge comments	Comments for bug/feature discussion and tracking
<b>Notifications</b>	Notifications for replies and new discussions	Notifications for PR reviews, approvals, merges	Notifications for updates, comments, or assignments

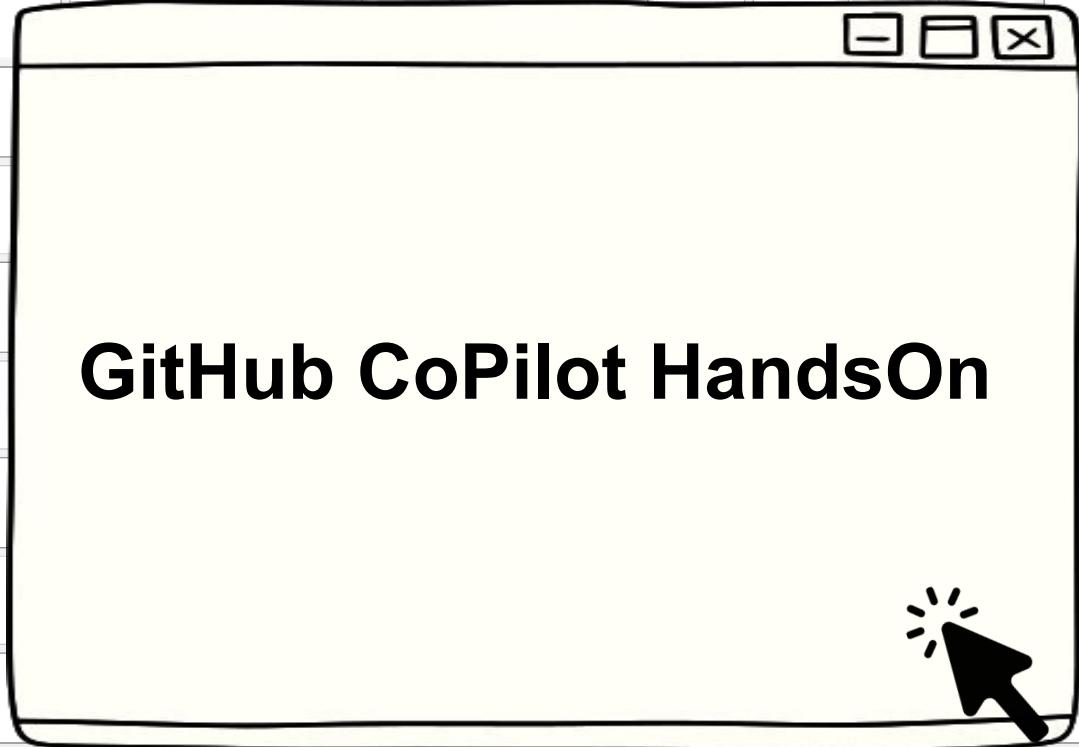


# Basic Syntax Formatting





Event Type	Description	Use Case
<b>push</b>	Triggered when commits are pushed to a branch.	Run tests or deploy after code is pushed.
<b>pull_request</b>	Triggered when a pull request is opened or updated.	Run CI tests before merging PR.
<b>workflow_dispatch</b>	Triggered manually through the GitHub UI.	Trigger workflows on demand (e.g., manual deploy).
<b>issue_comment</b>	Triggered when a comment is added to an issue.	Notify or take action based on issue comments.
<b>create</b>	Triggered when a new branch or tag is created.	Automatically create a release or deploy on tag creation.



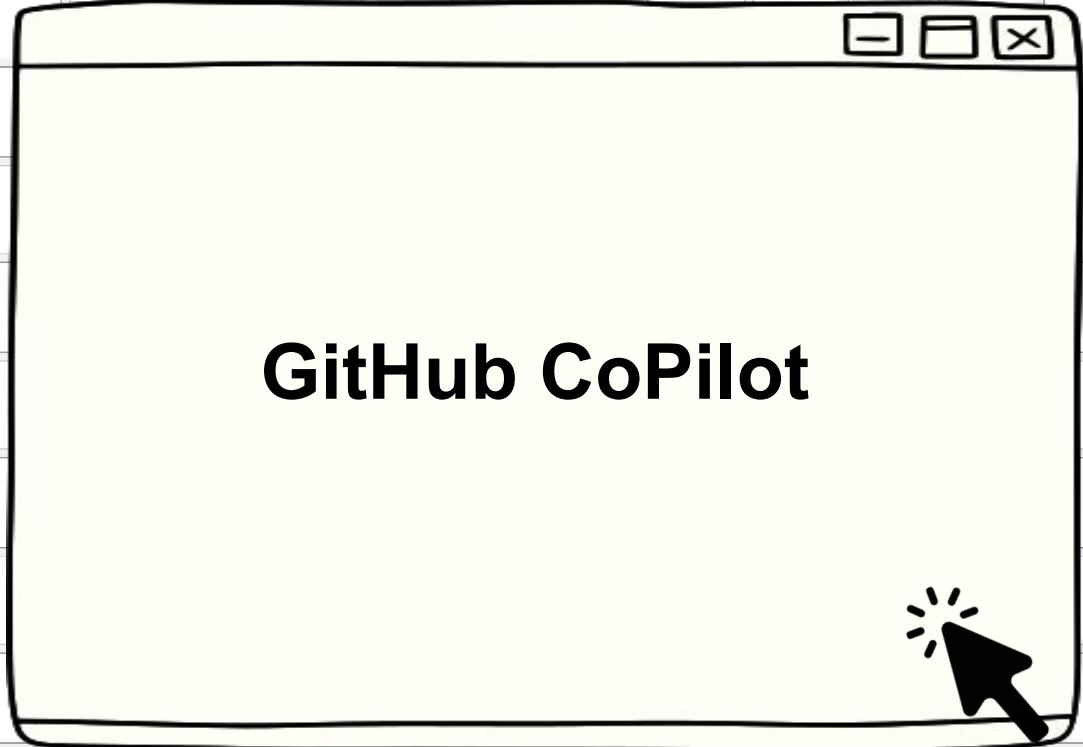




Karan Gupta



Feature	Copilot Free	Copilot Business
Who Can Use	Verified students, maintainers, individuals	Teams and organizations
Monthly Price	Free for eligible users, otherwise \$10	\$19 per user/month
Editor Support	VS Code, JetBrains, Neovim	Same as Free
Code Suggestions	Autocompletions, inline suggestions	Same, with admin controls
Access to Chat (Beta)	Available (for individuals)	Available



# GitHub CoPilot

- GitHub Copilot is an **AI-powered coding assistant** developed by GitHub and OpenAI.
- It helps developers write code faster by **suggesting entire lines, functions, or even files** as they type.

## Key Features:

- Autocompletes code in real-time
- Supports multiple languages (JavaScript, Python, Java, etc.)
- Works in editors like **VS Code, Neovim, JetBrains IDEs**
- Trained on public code from GitHub



## Use Cases:

- Writing boilerplate code
- Learning unfamiliar libraries
- Accelerating development

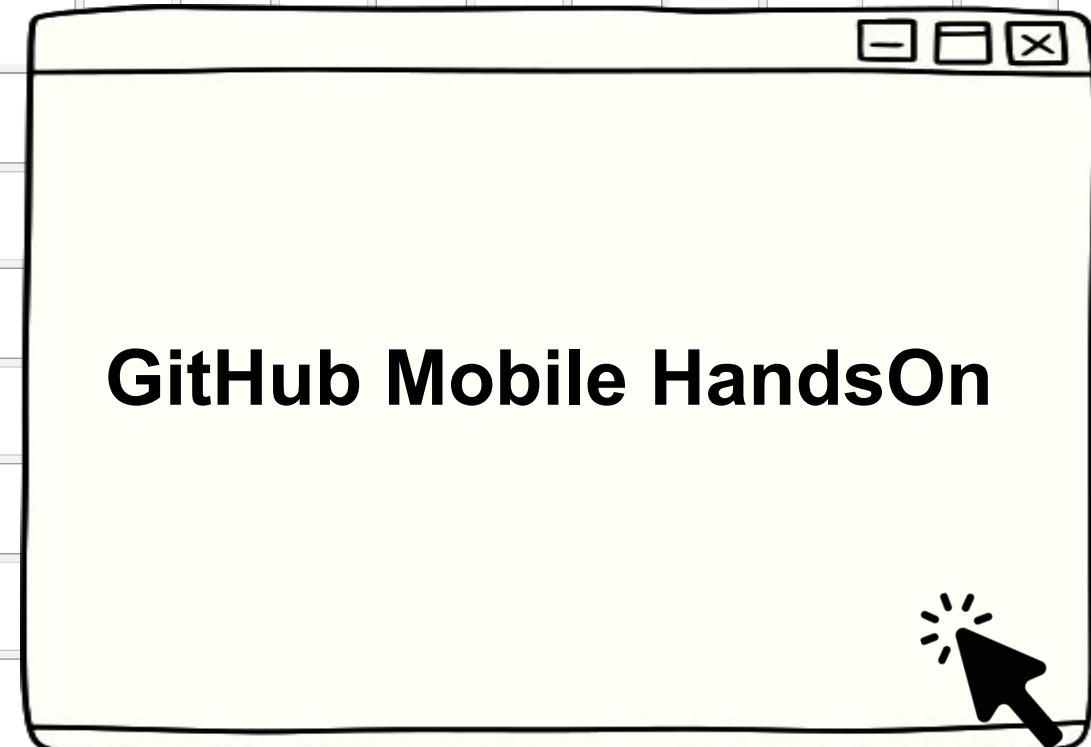
## Benefits:

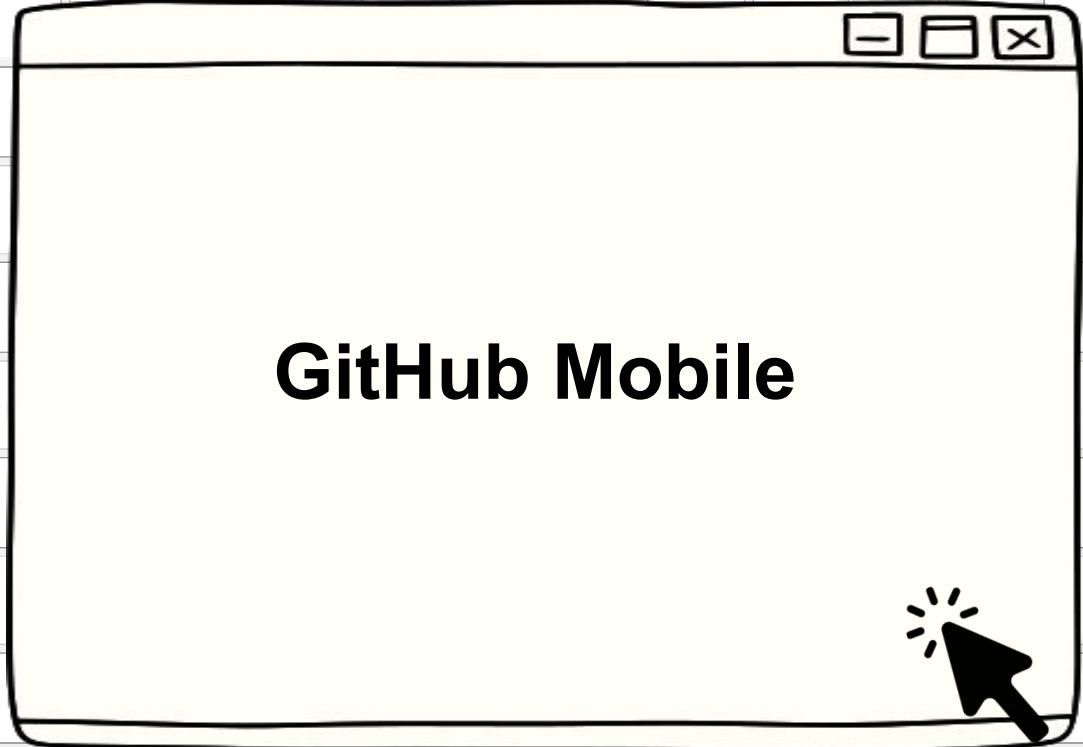
- Increases coding speed
- Boosts productivity
- Offers real-time learning for junior devs

## Limitations:

- May suggest incorrect or insecure code
- Always review before using in production







# GitHub Mobile

- GitHub Mobile is the **official mobile app** for GitHub, available on **iOS and Android**.
- It lets developers stay connected with their repositories **on the go**.

## **Key Features:**

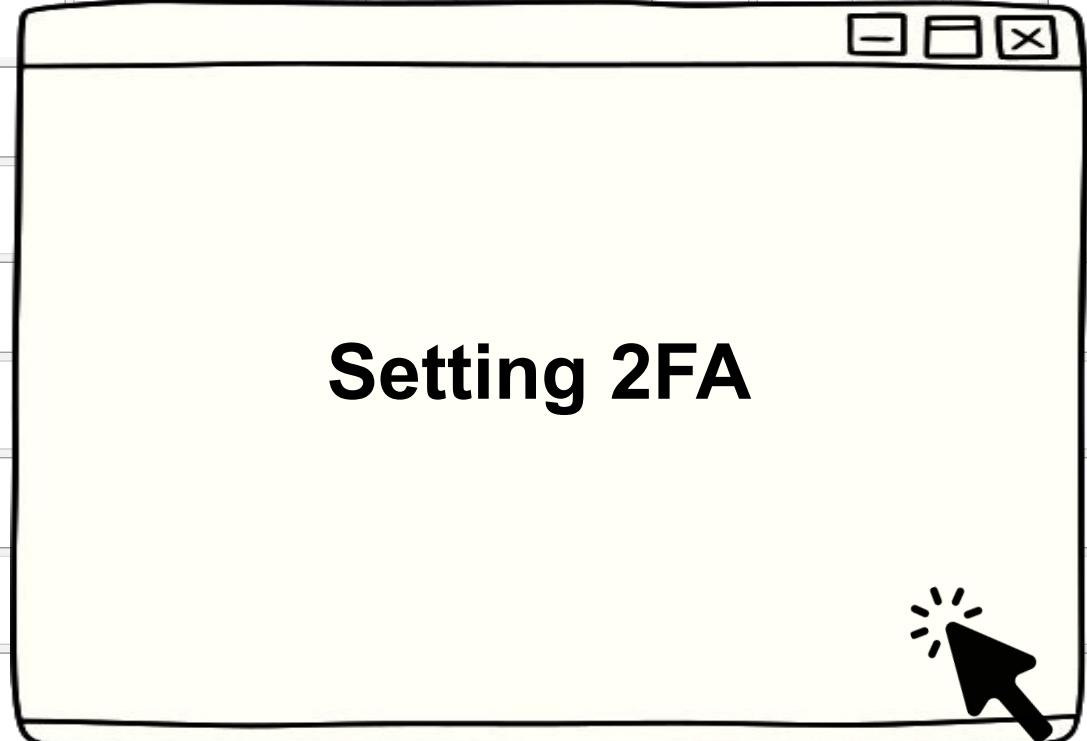
- **View repositories, pull requests, and issues**
- **Merge pull requests** directly from the app
- **Receive notifications** and respond quickly
- **Collaborate** with your team anytime, anywhere

## **Capabilities:**

- **Browse code** and files in repos
- **Comment on issues and PRs**
- **Approve/merge pull requests**
- **Push notifications** for updates
- **Dark mode** and responsive UI for comfort

## **Benefits:**

- Boosts productivity with **real-time collaboration**
- Helps you **stay on top of reviews** and issues
- Secure login with **biometrics support**



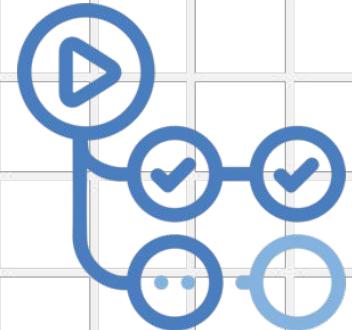


# GitHub Actions

- A built-in **CI/CD tool** in GitHub to **automate workflows** like build, test, and deployment.

## **Key Features:**

- Automates tasks based on GitHub events (e.g., push, PR)
- Supports **YAML-based workflow definitions**
- Access to **GitHub-hosted runners** (Linux, Windows, macOS)
- Integration with **marketplace actions** and custom scripts



## **Components of Workflow:**

<b>Component</b>	<b>Required</b>	<b>Description</b>
on	<input checked="" type="checkbox"/> Yes	Specifies event(s) that trigger the workflow
jobs	<input checked="" type="checkbox"/> Yes	Block defining jobs to run
runs-on	<input checked="" type="checkbox"/> Yes	Specifies the type of runner (e.g. ubuntu)
steps	<input checked="" type="checkbox"/> Yes	List of tasks to execute in the job
run / uses	<input checked="" type="checkbox"/> One is needed	Executes a script or action

```
1   name: Minimal CI Workflow
2
3   on: [push]
4
5   jobs:
6     example-job:
7       runs-on: ubuntu-latest
8
9     steps:
10      - name: Print Hello World
11        run: echo "Hello, World!"
```





# GitHub Notifications



- Alerts that keep you updated on activity in repositories you **watch, contribute to, or own**.

## **When Do You Get Notified?**

- New pull requests or issues
- Comments, mentions (@yourname)
- Review requests
- Changes to watched repositories or discussions

## **Notification Channels:**

- GitHub Web
- Email
- GitHub Mobile App



Karan Gupta

## Customization Options:

- **Watch settings:** Choose what you want to be notified about
- **Participating:** Notifications for threads you're involved in
- **Ignoring:** Turn off notifications for specific repos or threads

## How to Manage:

- Go to: [github.com/settings/notifications](https://github.com/settings/notifications)
- Set preferences for email, web, and mobile
- Use **Filters** and **Saved Replies** to manage responses





Karan Gupta



