DevOps

[Github 2](#_Toc194176217)

[Version Control System (VCS) 2](#_Toc194176218)

[Local and Central Version Control Systems 3](#_Toc194176219)

[Distributed Version Control System (DVCS) 5](#_Toc194176220)

[Git 7](#_Toc194176221)

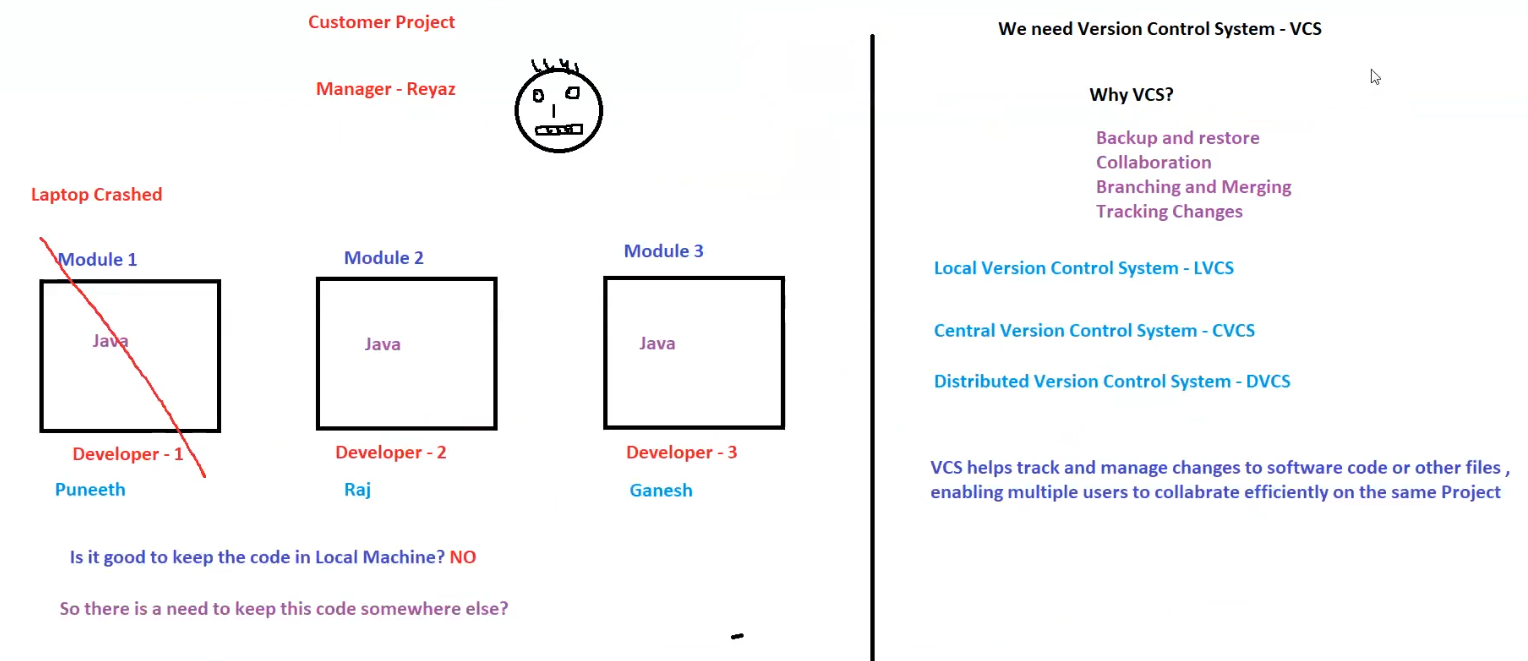
[Stages of Git 9](#_Toc194176222)

[Git workflow 11](#_Toc194176223)

[Git commands 13](#_Toc194176224)

# Github

## Version Control System (VCS)



**1. Scenario: The Need for Version Control System (VCS)**

* **Customer Project** managed by **Reyaz**.
* The project is divided into **three modules**, each developed by different developers:
  + **Module 1 (Java)** → Developer **Puneeth**
  + **Module 2 (Java)** → Developer **Raj**
  + **Module 3 (Java)** → Developer **Ganesh**

**Problem Identified**

* **Laptop Crash**: Developer **Puneeth’s laptop crashed**, leading to a potential loss of **Module 1** code.
* **Question Raised**:
  + **Is it good to keep code only on a local machine?** → **NO**
  + **Solution Needed**: A system to store and track code changes efficiently.

**2. Why Do We Need a Version Control System (VCS)?**

VCS provides essential functionalities to manage software development effectively:

1. **Backup and Restore** – Ensures that code is not lost due to system failures.
2. **Collaboration** – Enables multiple developers to work on the same project without conflicts.
3. **Branching and Merging** – Developers can work on different features separately and later merge changes.
4. **Tracking Changes** – Keeps a history of modifications, making it easier to debug and review updates.

**3. Types of Version Control Systems**

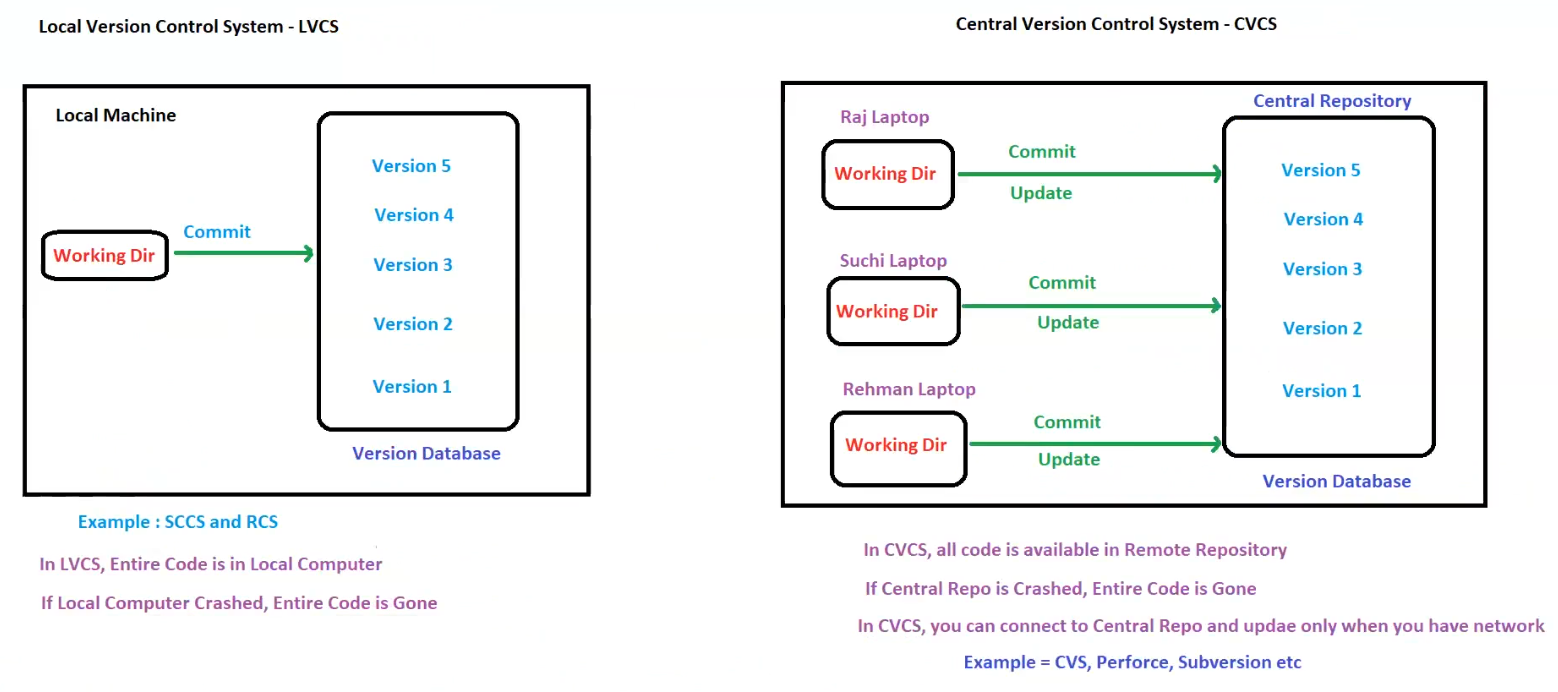
There are three main types of VCS:

1. **Local Version Control System (LVCS)**
   * Simple versioning method, typically managed locally on a developer's machine.
   * **Drawback**: If the local machine crashes, the code is lost.
2. **Central Version Control System (CVCS)**
   * A centralized server holds all versions of the code.
   * Examples: **Subversion (SVN), Perforce**
   * **Drawback**: If the central server crashes, access to all code versions is lost.
3. **Distributed Version Control System (DVCS)**
   * Each developer maintains a copy of the entire repository.
   * Examples: **Git, Mercurial**
   * **Advantage**: Even if the central server crashes, code exists on multiple developer machines.

**4. Conclusion: The Need for VCS**

* Storing code only on local machines is risky.
* **Solution**: Use a **Version Control System (VCS)** like **Git, GitHub, BitBucket, or GitLab**.
* **Benefit**: Enables **efficient collaboration**, **code safety**, and **seamless project management**.

## Local and Central Version Control Systems



**1. Local Version Control System (LVCS)**

**Definition**

* A **Local Version Control System (LVCS)** stores all code versions on a single **local machine**.
* Developers commit changes **locally**, and all versions are stored in a local database.

**How It Works**

* A **working directory** contains the latest version of the code.
* Developer commits changes, creating new **versions** (Version 1, 2, 3, etc.).
* The **version database** maintains all committed versions.

**Advantages**

* Simple to use.
* No internet connection required.

**Disadvantages**

* **Data loss risk**: If the local machine crashes, **all code is lost**.
* Not suitable for **team collaboration**.
* Difficult to track changes across multiple devices.

**Examples of LVCS**

* **SCCS (Source Code Control System)**
* **RCS (Revision Control System)**

**2. Central Version Control System (CVCS)**

**Definition**

* A **Central Version Control System (CVCS)** maintains all code in a **central repository**.
* Multiple developers can access, commit, and update changes from a **remote server**.

**How It Works**

* Developers (e.g., **Raj, Suchi, Rehman**) have **working directories** on their local machines.
* They **commit** changes to the **central repository**.
* They can also **update** their local copies by pulling the latest code.

**Advantages**

* **Collaboration**: Multiple developers can work on the same project.
* **Backup & Restore**: Code is stored remotely, reducing the risk of data loss from local failures.
* **Version tracking**: Enables tracking and rollback of changes.

**Disadvantages**

* **Single point of failure**: If the **central repository crashes**, all code versions may be lost.
* Requires **network connectivity** to access the latest updates.

**Examples of CVCS**

* **CVS (Concurrent Versions System)**
* **Perforce**
* **Subversion (SVN)**

**3. Key Differences Between LVCS and CVCS**

| **Feature** | **LVCS (Local VCS)** | **CVCS (Central VCS)** |
| --- | --- | --- |
| **Storage** | Code is stored **locally** on the developer's machine. | Code is stored in a **centralized repository**. |
| **Collaboration** | Not suitable for teams. | Multiple developers can work together. |
| **Risk of Data Loss** | **High** – If the machine crashes, code is lost. | **Moderate** – If the central server crashes, data is lost. |
| **Network Requirement** | **Not required** – Works entirely offline. | **Required** – Developers must connect to the central repository. |

**4. Conclusion**

* **LVCS** is useful for **individual developers** but has a high **data loss risk**.
* **CVCS** allows **team collaboration** and ensures **better data security**, but the **central server is a single point of failure**.
* The next step in version control is a **Distributed Version Control System (DVCS)** (e.g., **Git, Mercurial**), which provides **both local and remote copies** for better resilience.

## Distributed Version Control System (DVCS)

**1. What is DVCS (Distributed Version Control System)?**

* A **Distributed Version Control System (DVCS)** allows every developer to have a **full copy** of the repository.
* The entire codebase and history are **stored both locally and in a remote repository**.
* Developers can work offline and sync changes only when needed.

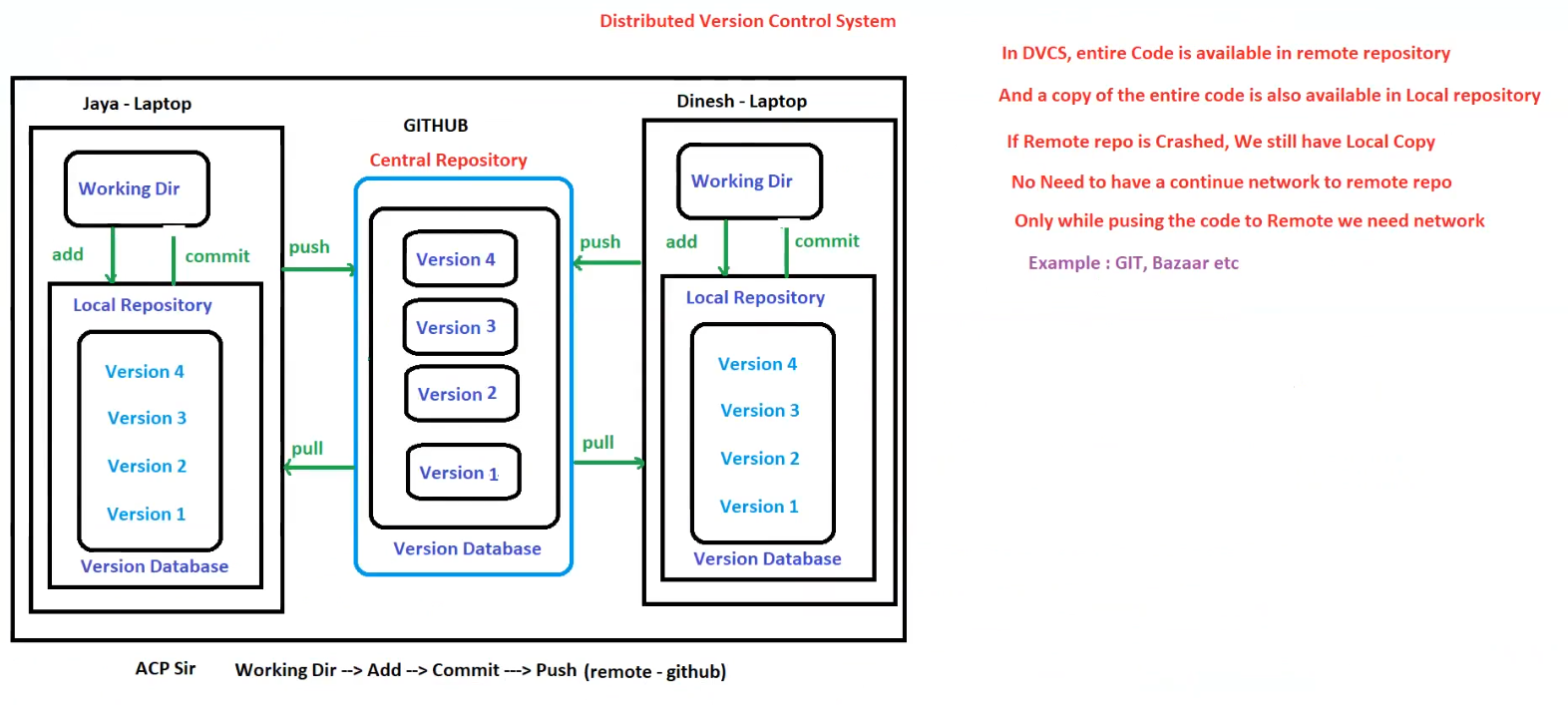
**2. How DVCS Works**

**Components**

1. **Working Directory** – The developer's workspace where changes are made.
2. **Local Repository** – A local version database storing all previous versions.
3. **Central Repository (GitHub, GitLab, BitBucket, etc.)** – The remote repository for collaboration.

**Workflow**

1. **Add** – The developer stages changes from the **working directory**.
2. **Commit** – The staged changes are saved into the **local repository**.
3. **Push** – The committed changes are uploaded to the **central repository** (e.g., GitHub).
4. **Pull** – The developer fetches the latest changes from the **central repository** to their local repository.



**3. Advantages of DVCS**

✅ **Local Backup** – If the remote repository crashes, developers still have a local copy.  
✅ **Work Offline** – No need for a continuous network connection; developers can commit changes locally.  
✅ **Faster Operations** – Since commits and version history are available locally, performance is improved.  
✅ **Better Collaboration** – Multiple developers can work on separate branches and merge changes easily.

**4. Key Differences Between CVCS and DVCS**

| **Feature** | **CVCS (Centralized)** | **DVCS (Distributed)** |
| --- | --- | --- |
| **Storage** | Code is stored in a **centralized repository**. | Every developer has a **local copy of the entire repo**. |
| **Work Offline** | No, requires **network access**. | Yes, developers can work offline. |
| **Risk of Data Loss** | High – If the **central server crashes**, all code versions may be lost. | Low – If the central server crashes, local copies still exist. |
| **Examples** | CVS, SVN, Perforce | **Git, Bazaar, Mercurial** |

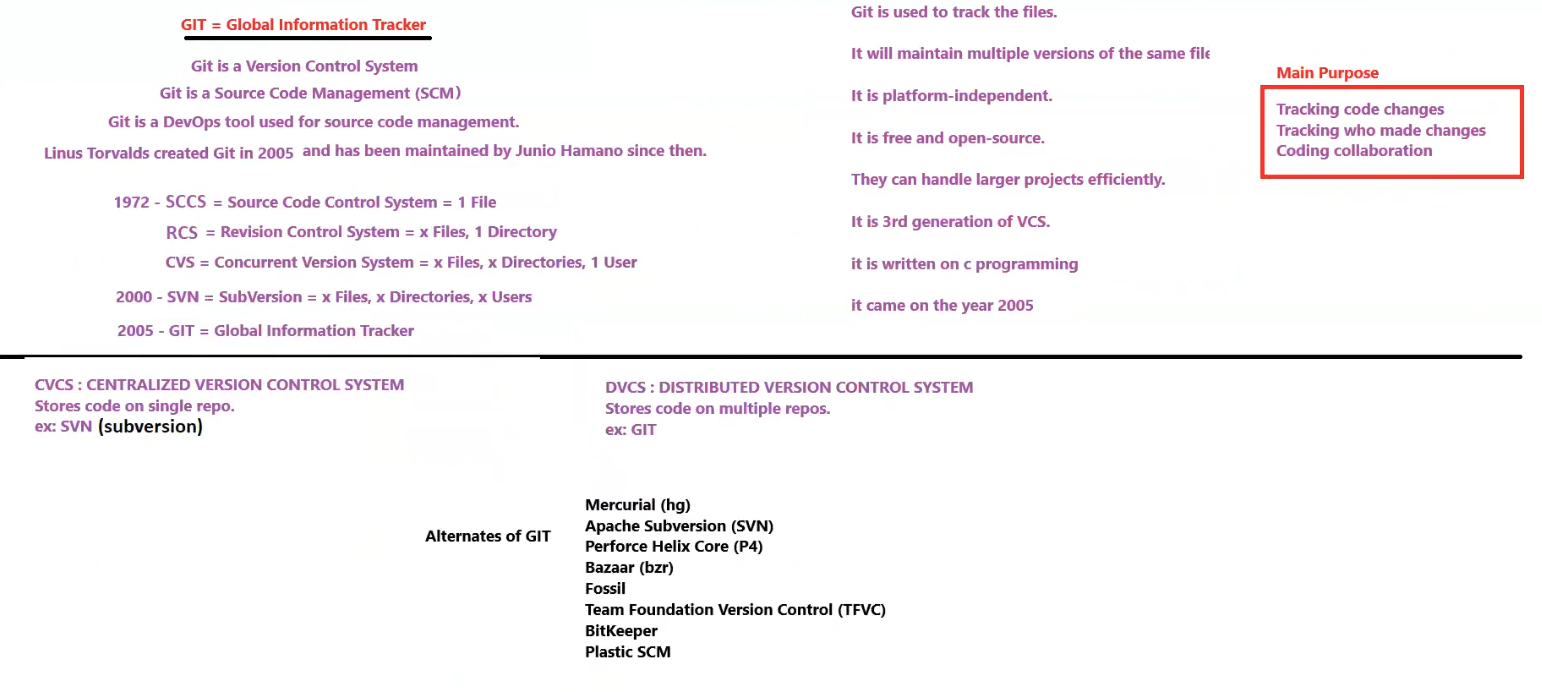
**5. Example Tools for DVCS**

* **Git** (Most popular, used with GitHub, GitLab, BitBucket)
* **Bazaar**
* **Mercurial**

**6. Conclusion**

* **DVCS is more reliable and flexible** than CVCS.
* **Git is the most widely used DVCS** today.
* Developers can work **offline**, **collaborate efficiently**, and **avoid data loss**.

## Git



**1. What is Git?**

* **Git = Global Information Tracker**
* Git is a **Version Control System (VCS)** that helps track changes in code.
* It is also a **Source Code Management (SCM)** tool, widely used in **DevOps**.
* **Created by Linus Torvalds in 2005**, and later maintained by Junio Hamano.

**2. Evolution of Version Control Systems**

| **Year** | **System** | **Description** |
| --- | --- | --- |
| **1972** | SCCS (Source Code Control System) | Managed single files. |
| **1980s** | RCS (Revision Control System) | Managed multiple files in one directory. |
| **1990s** | CVS (Concurrent Version System) | Allowed multiple users and directories. |
| **2000** | SVN (Subversion) | Improved upon CVS with better directory tracking. |
| **2005** | Git | Fully **distributed**, supports offline work, and is fast. |

**3. Key Features of Git**

✅ **Tracks code changes efficiently.**  
✅ **Maintains multiple versions of a file.**  
✅ **Platform-independent, free, and open-source.**  
✅ **Handles large projects efficiently.**  
✅ **Written in C for performance optimization.**  
✅ **Third-generation version control system.**

**4. Centralized vs. Distributed Version Control Systems**

| **Feature** | **CVCS (Centralized VCS)** | **DVCS (Distributed VCS)** |
| --- | --- | --- |
| **Storage** | Single central repository. | Multiple copies across different systems. |
| **Example** | SVN (Subversion) | Git, Mercurial, Bazaar |
| **Network Dependency** | Requires network access for most operations. | Can work **offline**, network needed only for push/pull. |
| **Failure Risk** | If the central server crashes, data can be lost. | No risk, as every developer has a full copy. |

**5. Main Purpose of Git**

📌 **Tracking code changes.**  
📌 **Tracking who made changes.**  
📌 **Facilitating team collaboration in coding.**

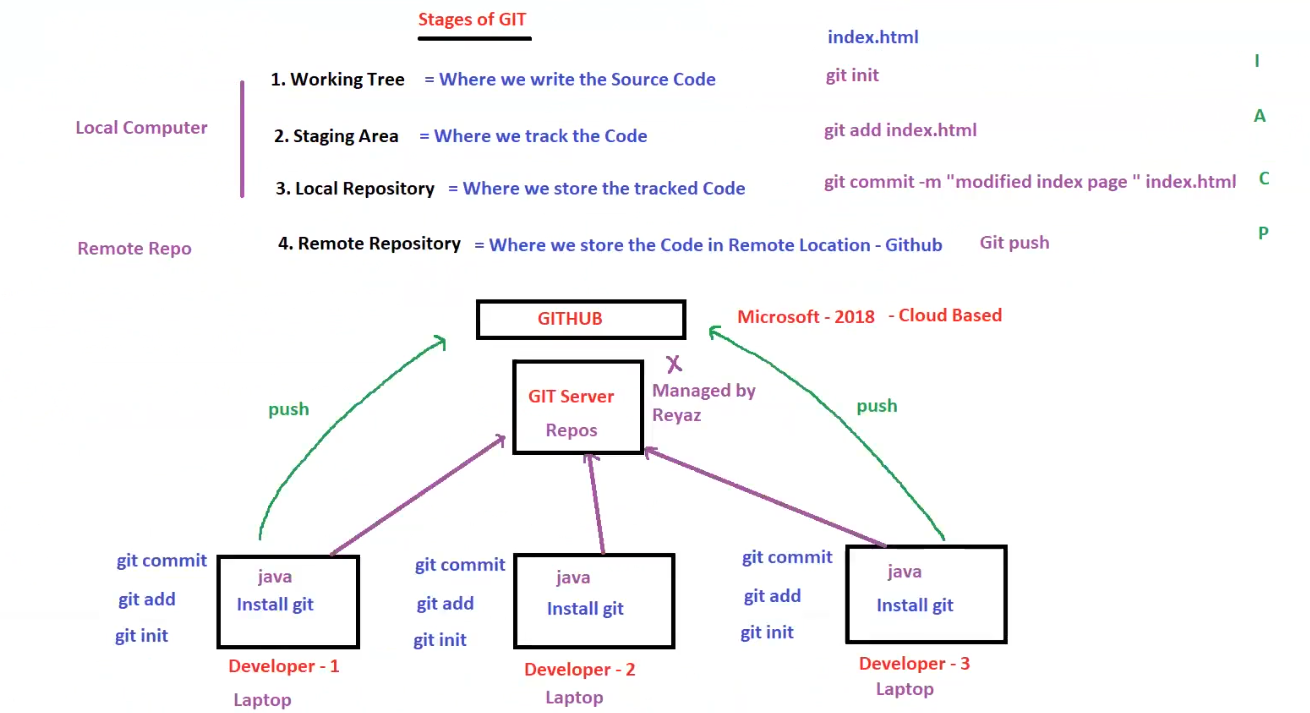
**6. Alternatives to Git**

* **Mercurial (hg)**
* **Apache Subversion (SVN)**
* **Perforce Helix Core (P4)**
* **Bazaar (bzr)**
* **Fossil**
* **Team Foundation Version Control (TFVC)**
* **BitKeeper**
* **Plastic SCM**

**Conclusion**

Git has become the most widely used **Distributed Version Control System (DVCS)** due to its speed, offline capability, and efficient tracking features. 🚀

## Stages of Git



**1. Stages of Git**

Git follows a **four-stage** process for version control:

**1️⃣ Working Tree**

* This is where developers **write and modify source code**.
* Files in this stage are **untracked** until they are added to Git.
* **Example:**
* git init # Initialize a new Git repository

**2️⃣ Staging Area**

* This is where we **track changes** before committing them.
* Files must be **added** to the staging area before committing.
* **Example:**
* git add index.html # Track changes to index.html

**3️⃣ Local Repository**

* This is where we **store the tracked code** locally after committing.
* Commits create a **snapshot** of the project.
* **Example:**
* git commit -m "Modified index page"

**4️⃣ Remote Repository**

* The remote repository (e.g., **GitHub**) stores the code **in a remote location**.
* Developers push their local commits to the remote repository.
* **Example:**
* git push origin main

**Mnemonic to Remember Git Stages:**

📌 **IAC P** → **Init, Add, Commit, Push**

**2. GitHub and Remote Repositories**

* **GitHub** is used as a remote repository.
* GitHub was **acquired by Microsoft in 2018** and is now **cloud-based**.
* Multiple developers push their commits to the **Git Server** for collaboration.

**3. Git Workflow for Multiple Developers**

**Workflow Steps**

1. **Developers initialize Git on their local machines**
2. git init
3. **Developers add and commit changes locally**
4. git add .
5. git commit -m "Initial commit"
6. **Developers push code to the shared remote repository**
7. git push origin main
8. **Other developers can pull the latest changes**
9. git pull origin main

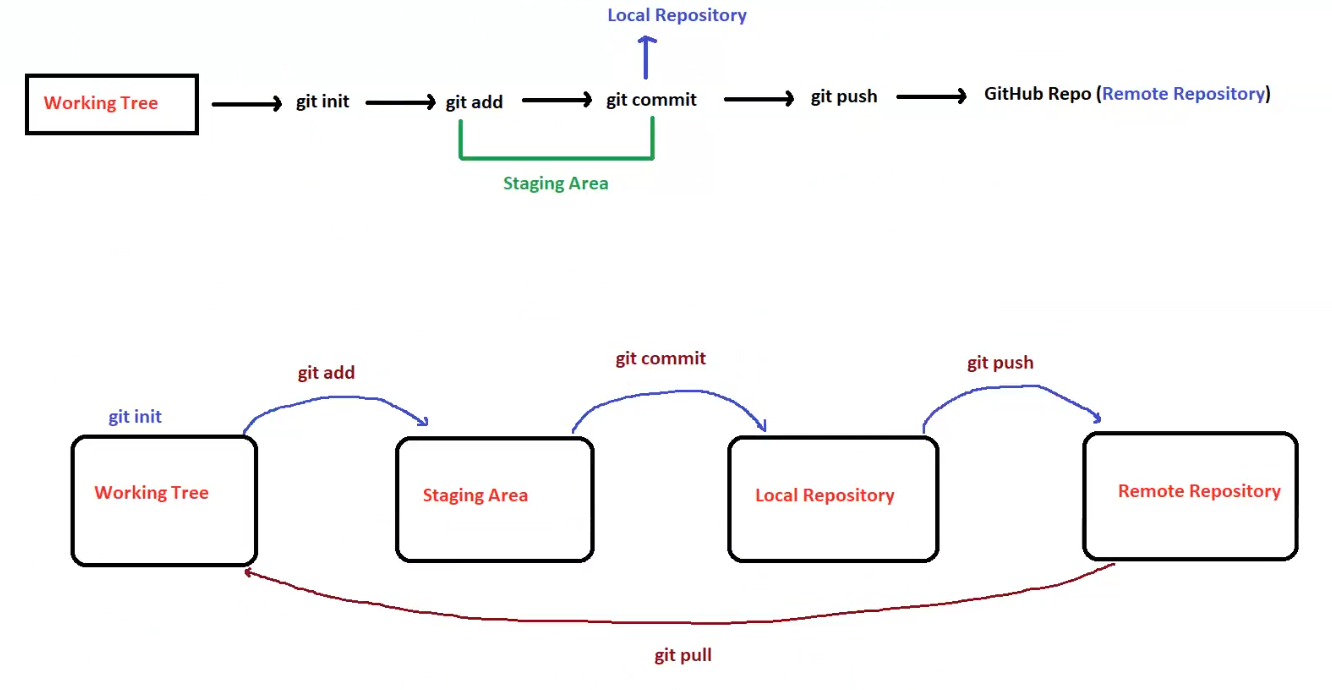
**Collaboration in a Git Repository**

* **Developer 1, Developer 2, and Developer 3** each have **local repositories**.
* All developers **push** changes to the shared **GitHub repository**.

**4. Key Takeaways**

✅ **Git has four main stages: Working Tree, Staging Area, Local Repo, and Remote Repo.**  
✅ **GitHub serves as a cloud-based remote repository for collaboration.**  
✅ **Git allows multiple developers to work on the same project efficiently.**

## Git workflow



**🌳 1. Working Tree**

* The **Working Tree** is the directory where you make changes to your project (e.g., adding, editing, or deleting files).
* This is where you write code or make modifications.

✅ **Command:**

bash

CopyEdit

git init

* Initializes a new Git repository in the current directory.
* Creates a .git folder that contains all the Git metadata.

**📥 2. Staging Area**

* When you’re ready to save your changes, you add them to the **Staging Area**.
* The staging area is like a holding area where you list the changes you want to include in the next commit.

✅ **Command:**

bash

CopyEdit

git add <file\_name>

# or to add all changes:

git add .

* Moves the changes from the working directory to the staging area.
* Files in this area are marked as ready to be committed.

**🔒 3. Local Repository**

* After staging, you create a snapshot of the current state by committing the changes to the **Local Repository**.
* This is where Git permanently stores the history of your project (on your machine).

✅ **Command:**

bash

CopyEdit

git commit -m "Commit message"

* Saves the staged changes with a message describing what was modified.

**🚀 4. Remote Repository (GitHub)**

* The remote repository is a version of your project stored on a server, such as GitHub, GitLab, or Bitbucket.
* Pushing the code makes your local changes available to others.

✅ **Command:**

bash

CopyEdit

git push origin <branch\_name>

* Sends the committed changes from the local repository to the remote repository.

**🔄 5. Git Pull**

* To get the latest changes from the remote repository, you pull them into your local repository.
* This keeps your local repository in sync with the remote repository.

✅ **Command:**

bash

CopyEdit

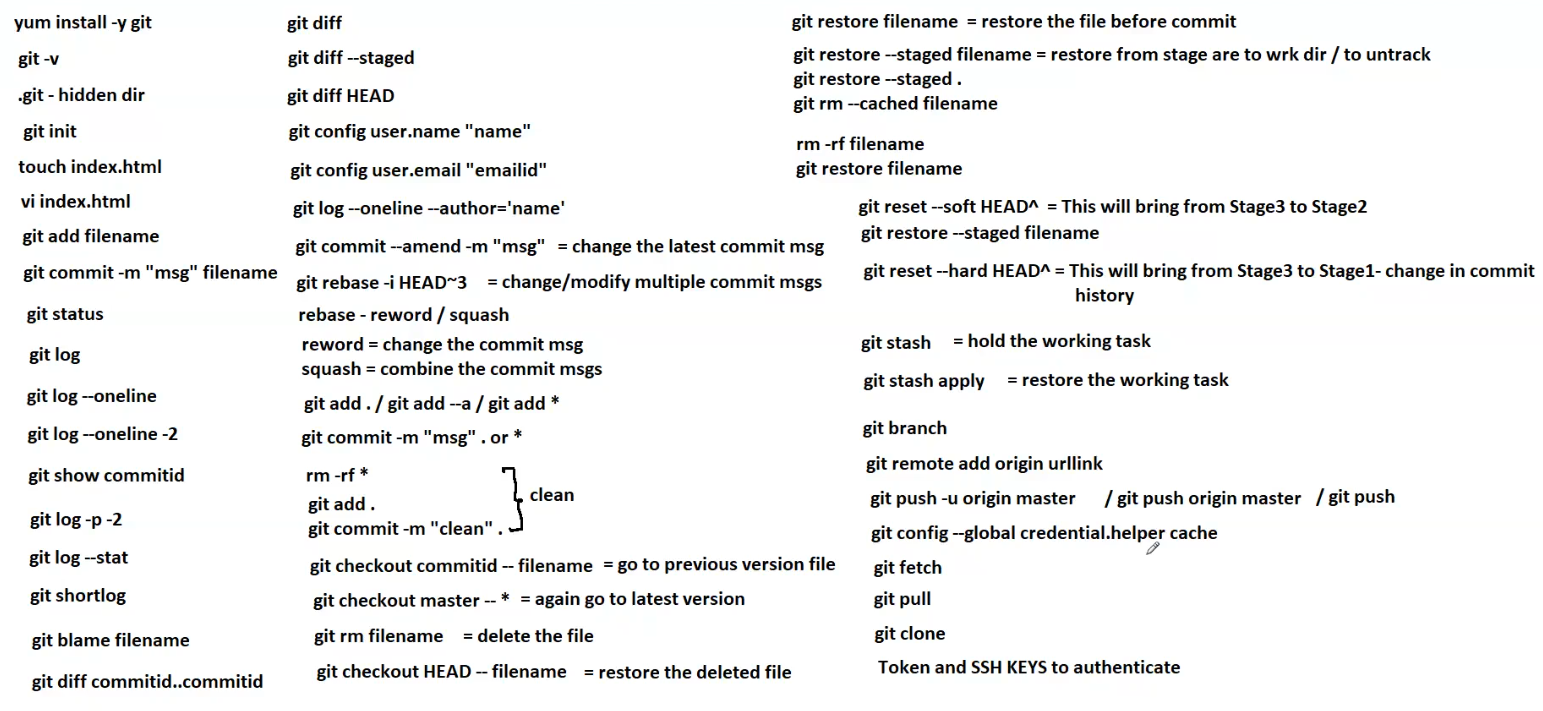
git pull origin <branch\_name>

* Merges remote changes into your local repository.

**🔁 Full Workflow Summary**

1. **Initialize Git:**  
   git init → Creates a new local repository.
2. **Add Files to Staging Area:**  
   git add <file\_name> → Prepares files for commit.
3. **Commit Changes:**  
   git commit -m "message" → Saves the snapshot to the local repository.
4. **Push to Remote Repository:**  
   git push origin <branch\_name> → Uploads changes to GitHub.
5. **Pull Changes from Remote:**  
   git pull origin <branch\_name> → Syncs local repository with remote changes.

## Git commands





**✅ 1. Git Installation and Initialization**

* **Install Git:**
* yum install -y git
* **Check Git version:**
* git -v
* **Initialize a Git repository:**
* git init
* **Check for hidden .git directory:**
* ls -a # reveals .git hidden dir

**🛠️ 2. Basic File and Directory Operations**

* **Create a new file:**
* touch index.html # create an empty file
* **Edit a file using vim editor:**
* vi index.html

**📥 3. Staging and Committing**

* **Add a file to the staging area:**
* git add filename
* **Commit the changes with a message:**
* git commit -m "your message" filename

**🔍 4. Viewing the Repository Status and Logs**

* **Check the status of the working directory:**
* git status
* **View the commit history:**
* git log
* **Simplified log view:**
* git log --oneline
* **View the last 2 commits in oneline format:**
* git log --oneline -2
* **Show a specific commit ID:**
* git show <commit\_id>
* **View patch/diff details for the last 2 commits:**
* git log -p -2
* **View log statistics:**
* git log --stat
* **View short log format:**
* git shortlog
* **Check blame (who last modified each line):**
* git blame filename

**🔁 5. Comparing and Differentiating**

* **Show differences between current and last commit:**
* git diff
* **Show staged differences:**
* git diff --staged
* **Compare HEAD with the working directory:**
* git diff HEAD
* **Compare two specific commits:**
* git diff commitid1..commitid2

**🔧 6. Configuration and User Info**

* **Set username:**
* git config user.name "Your Name"
* **Set email:**
* git config user.email "youremail@example.com"
* **View logs by a specific author:**
* git log --oneline --author='name'

**🔨 7. Amend and Modify Commits**

* **Change the latest commit message:**
* git commit --amend -m "new message"
* **Modify multiple previous commit messages:**
* git rebase -i HEAD~3 # modify the last 3 commits
* **Options during rebase:**
  + reword → Change the commit message.
  + squash → Combine multiple commits into one.

**🔥 8. Adding and Cleaning Up**

* **Add all changes:**
* git add .
* git add -A
* git add \*
* **Commit with a general message:**
* git commit -m "message" .
* **Remove all files forcefully:**
* rm -rf \*
* **Add, commit, and clean:**
* git add .
* git commit -m "clean"

**🔁 9. Checkout and Restore**

* **Go to a previous version of a file:**
* git checkout <commitid> -- filename
* **Return to the latest version:**
* git checkout master -- \*
* **Delete a file:**
* git rm filename
* **Restore a deleted file:**
* git checkout HEAD -- filename

**🔧 10. Undo and Reset Changes**

* **Restore a file before commit:**
* git restore filename
* **Unstage a file (move from staging to working directory):**
* git restore --staged filename
* **Unstage all files:**
* git restore --staged .
* **Remove a file from staging:**
* git rm --cached filename
* **Remove a file forcefully:**
* rm -rf filename

**🚀 11. Reset and Stash**

* **Reset to previous stage (soft):**
* git reset --soft HEAD^
  + Moves from Stage 3 to Stage 2 (keeps changes staged).
* **Reset to previous stage (hard):**
* git reset --hard HEAD^
  + Moves from Stage 3 to Stage 1 (discards changes).
* **Stash current changes:**
* git stash
* **Restore stashed changes:**
* git stash apply

**🌐 12. Branching and Remote**

* **Create and manage branches:**
* git branch
* **Add a remote repository:**
* git remote add origin <repository\_url>
* **Push to remote repository:**
* git push -u origin master
  + Alternate commands:
* git push origin master
* git push

**🔑 13. Authentication and Credentials**

* **Cache credentials globally:**
* git config --global credential.helper cache
* **Token and SSH keys authentication:**
  + Use personal access tokens (PAT) or SSH keys for secure authentication.

**🔄 14. Fetching, Pulling, and Cloning**

* **Fetch the latest changes:**
* git fetch
* **Pull the latest changes:**
* git pull
* **Clone a repository:**
* git clone <repository\_url>

**🌿 15. Branch Management**

* **List all branches:**
* git branch
* **Create a new branch:**
* git branch <branchname>
* **Switch to an existing branch:**
* git checkout <branchname>
* **Create and switch to a new branch:**
* git checkout -b <branchname>
* **Rename the current branch:**
* git branch -m <currentbranch> <newbranch>
* **Delete a branch forcefully:**
* git branch -D <branchname>

**🔀 16. Merging and Rebasing**

* **Merge another branch into the current one:**
* git merge <branchname>
* **Revert the changes from a branch:**
* git revert <branchname>
* **Rebase the current branch onto another branch:**
* git rebase <branchname>

**✅ 17. Committing and Cleaning**

* **Commit with a message and stage all changes:**
* git commit -am "your message"
* **Force clean untracked files:**
* git clean -f <filename>

**🍒 18. Cherry-picking and Tagging**

* **Cherry-pick a specific commit (apply changes from a specific commit to the current branch):**
* git cherry-pick <commitid>
* **Tag a specific commit:**
* git tag <tagname> <commitid>

**🛠️ 19. .gitignore and RefLog**

* **Open and edit the .gitignore file:**
* vi .gitignore
* **View the reference log (history of HEAD changes):**
* git reflog

**🔥 20. Checkout with Specific Commit**

* **Create a new branch from a specific commit:**
* git checkout -b <branchname> <commitid>