

# Version Control Systems Overview

- **Version Control System (VCS)**

A tool that tracks changes to files over time, enabling multiple developers to collaborate and maintain historical versions of a codebase.

- **Why VCS became essential**
    - Manages **code sharing** among team members.
    - Provides robust **versioning** to revert, compare, or audit changes.
- 

## Problems Solved by VCS

### **1** Sharing Code

- In a team (e.g., **Dev 1** and **Dev 2**) working on a *calculator* app:
  - Dev 1 implements **addition**, Dev 2 implements **subtraction**.
  - Both pieces must be merged into a single, functional application.
- Simple file-email or chat sharing works for tiny projects, but **real-world** applications involve:
  - **Hundreds of packages, thousands of files.**
  - Complex **dependency trees** (e.g., JAR files).
- VCS automates merging, conflict detection, and ensures every developer has the latest integrated code.

### **2** Versioning (History & Rollback)

- Feature evolution example:
    - Start: **addition of two numbers**.
    - Change: expand to **addition of three numbers**, then **four numbers**.
    - Later decision: revert to **addition of two numbers** only.
  - Without VCS, tracking which code corresponds to which requirement (day 1, day 2, ...) is impractical.
  - VCS records each **commit** as a distinct version, allowing:
    - Retrieval of any past state (e.g., 10, 20, 50 days ago).
    - Auditing of who changed what and why.
- 

## Centralized vs. Distributed VCS

Feature	Centralized VCS (e.g., SVN)	Distributed VCS (e.g., Git)
Repository location	Single <b>central server</b>	Every developer has a <b>full copy</b>
Workflow	Developers push/pull directly to/from the central server	Developers work locally, then push changes to any remote (or share via their own copy)
Failure tolerance	If the central server goes down, collaboration halts	Work can continue offline; any copy can act as a new source
Branching & merging	Typically heavier, slower	Lightweight, fast, encourages frequent branching
Typical use case	Small teams, simple projects	Large, distributed teams, open-source contributions

**Interview Cue:** “What is the difference between a **centralized** and a **distributed** version control system?” – focus on repository location, fault tolerance, and workflow.

## Git and Forking


- **Git** implements a **distributed** model: each clone contains the entire repository history.
- **Fork**

The act of creating a **complete personal copy** of an existing repository (e.g., fork-Mahadev from example.com).

- Benefits of forking:
  - Guarantees an independent backup of the whole codebase.
  - Enables developers to work in isolation and later propose changes via pull requests.
  - Removes reliance on a single central server—if the original goes down, the fork remains functional.

- Typical scenario:
    1. Original repository: example.com/repo.git.
    2. Developer **Mahadev** creates **Fork Mahadev**.
    3. Mahadev makes changes locally, commits, and can push to his fork.
    4. When ready, he can request integration back to the original repository.
- 

## Key Takeaways

- **Version control** solves **sharing** and **versioning** challenges in collaborative software development.
  - **Centralized VCS** relies on a single server; **distributed VCS** (Git) provides copies for each participant, enhancing resilience and flexibility.
  - **Forking** is the core mechanism that makes Git truly distributed, allowing developers to maintain independent, full-history copies of a project. ## 
- Centralized vs Distributed Version Control Systems

**Centralized VCS:** A single server stores the definitive repository; developers must be online to pull or push changes.

- **Problem with SVN/CVS**
    - Single point of failure → if the server goes down, *Developer A* and *Developer B* cannot exchange code.
    - System administrators managed the repository on a Linux machine; downtime meant **no communication** between developers.
  - **Distributed VCS (DVCS)**
    - Every clone contains the full history, so work can continue offline.
- 

## Git vs. GitHub (and similar services)

**Git:** An open-source **distributed version control system** that can be installed on any server (e.g., an EC2 instance).

**GitHub / GitLab / Bitbucket:** Hosted platforms that **wrap Git** with additional features such as issue tracking, code review, and project management.

- Organizations can run their own Git server → raw Git provides only versioning.
  - Hosted services add:
    - Pull-request workflow
    - Inline comments & code reviews
    - Integrated CI/CD pipelines
    - Project boards & milestone tracking
- 

## Creating a Git Repository

### 1. Install Git

- Download from the official site ([git-scm.com/download](https://git-scm.com/download)).
- Choose the appropriate package for your OS (Linux, macOS, Windows).
- Verify installation with `git --version`.

### 2. Initialize a repository

```
git init
```

- Creates an empty repository and a hidden `.git` directory.

### 3. Add files (e.g., `calculator.sh`).

### 4. Commit the initial version (see workflow below).

---

## Inside the .git Directory

Folder / File	Purpose
<code>objects/</code>	Stores all <b>objects</b> (blobs, trees, commits) that represent file contents and history.
<code>refs/</code>	Contains pointers to heads of branches, tags, and other references.
<code>hooks/</code>	Scripts that run at key points (e.g., pre-commit) to enforce policies (e.g., prevent committing secrets).

<b>config</b>	Repository-level configuration (user name, email, remote URLs, credential helpers).
<b>HEAD</b>	Points to the current branch tip; essential for determining the working tree state.



## Basic Git Workflow

The **Git lifecycle** that every developer or DevOps engineer should master.

### 1 Track Changes

```
git status
```

- Shows **untracked** files (e.g., calculator.sh).

```
git add <file>
```

- Moves the file to the *staging area*; Git now knows to track it.

### 2 Record a Snapshot

```
git commit -m "Add calculator script"
```

- Creates a new **commit object** representing the current state of the staged files.

### 3 Share with Others

```
git push <remote> <branch>
```

- Sends local commits to a remote repository (e.g., GitHub).

### 4 Review Modifications

## git diff

- Shows line-by-line changes between the working tree and the last commit (or between any two commits).

### 5 Repeat

- After editing calculator.sh (e.g., adding a third operand), run git status → see **modified** file.
- Use git diff to inspect the exact change (+ b + c).
- git add, git commit, and git push to advance the version history.

## Common Git Commands Cheat Sheet

Command	Description
git init	Create a new empty repository.
git clone <url>	Copy an existing remote repo locally.
git status	View current tracking status.
git add <file>	Stage a file for the next commit.
git commit -m "msg"	Record staged changes with a message.
git push <remote> <branch>	Upload commits to a remote.
git pull	Fetch and merge changes from a remote.
git diff	Show differences between versions.
git log	Browse commit history.

- **Commit** – a snapshot of the project stored in the repository.

**Commit:** *A record that captures the state of the files at a specific point, identified by a unique SHA-1 hash.*

- **Branch** – a movable pointer to a commit, typically used to develop features independently.

**Branch:** *An independent line of development that starts from a commit and can diverge from other lines.*

Typical workflow after editing a file:

1. **Check status** – see what Git knows about the working tree.

```
git status
```

2. **Stage changes** – tell Git which modifications should be included in the next commit.

```
git add calculator.sh
```

3. **Commit** – record the staged changes with a descriptive message.

```
git commit -m "first version of addition"
```

Git reports messages such as “nothing to commit, working tree clean” or “one untracked file” to keep you aware of the repository state.



## Making Changes, Inspecting, and Committing

When you add a new feature (e.g., subtraction) follow the same cycle:

1. Edit calculator.sh (e.g., change  $Y = A - B$ ).
2. Verify the modification:

```
git status
```

3. Review **exact** differences before staging:

```
git diff
```

4. Stage the file:

```
git add calculator.sh
```

5. Commit with a new message:

```
git commit -m "second version with subtraction"
```

**Tip:** Run `git status` after each step to know the next required command.

---



## Viewing History

Command	What it Shows
<code>git log</code>	List of commits (ID, author, message)
<code>git show &lt;commit-id&gt;</code>	Details of a specific commit
<code>git diff &lt;commit1&gt; &lt;commit2&gt;</code>	Differences between two commits

The log displays entries such as:

- Commit 1 – *author: Mahadev* – “first version of addition”
- Commit 2 – *author: Mahadev* – “second version with subtraction”

Each entry includes a unique **commit ID** (SHA-1 hash) used for later operations.

---



## Reverting to a Previous Version

1. Find the desired commit ID with `git log`.
2. Reset the repository to that commit (hard reset discards later changes):

```
git reset --hard <commit-id>
```

3. Verify the file content:



```
cat calculator.sh
```

The output should reflect the state of the chosen commit (e.g., only the addition line).

---

## Sharing Code – Remote Repositories

Git is a **distributed version-control system**. To collaborate, you push a local repository to a remote service such as **GitHub**, **GitLab**, or a **self-hosted Git server**.

### Creating a Remote Repository (GitHub example)

1. **Sign up** on github.com (email → answer questions → create account).
2. Click the **New** button to create a repository.
3. Fill in details:
  - **Repository name** (e.g., calculator-shell).
  - **Visibility** – *Public* (anyone can view) or *Private* (restricted access).
  - **Initialize** with a README (provides metadata about the project).
4. Click **Create repository**.

### Repository Visibility Comparison

Visibility	Who Can See the Code	Typical Use Case
<b>Public</b>	Anyone on the internet	Open-source projects
<b>Private</b>	Only invited collaborators (or organization members)	Proprietary or internal projects

After the remote repository exists, you can **push** local commits to it (the lecture mentions this conceptually; specific commands were not listed).

---

## Collaborating with Others

- **Fork** – creates a personal copy of a public repository, allowing you to experiment without affecting the original.
- **Pull request** – once changes are ready, you propose them to be merged back into the upstream repository.

These mechanisms enable multiple developers to work on the same codebase while keeping each version history intact.## 🌐 What is **GitHub**

**GitHub** is a cloud-based platform that hosts Git repositories and provides collaboration tools such as issue tracking, pull requests, and integrated CI/CD pipelines.

## 🔧 Why **GitHub** Is So Popular

- **Community size** – millions of developers share and discover code.
- **Rich ecosystem** – extensive marketplace of actions, integrations, and apps.
- **User-friendly UI** – intuitive web interface for managing repos, discussions, and documentation.
- **Built-in CI/CD** – native **GitHub Actions** simplify automation without external services.
- **Robust security** – features like Dependabot, secret scanning, and role-based access control.

*Compared to **GitLab** and **Bitbucket**, GitHub often wins on network effects and third-party integrations, while self-hosted Git solutions excel in full on-prem control.*

## 🏢 Managing **Users** & **Organizations**

- **Users**: personal accounts that can own repositories, fork projects, and contribute via pull requests.
- **Organizations**: groups that own multiple repositories, enable team-based permission management, and support shared billing.

Entity	Typical Use Case	Permissions Example
<b>User</b>	Individual developer or hobbyist	Owner of personal repos, collaborator
<b>Org Team</b>	Development squad, QA, design, ops	Read, write, admin scoped to specific repos
<b>Org</b>	Company or open-source community	Centralized billing, enterprise policies

## 🔧 Core Features

## **Issues – Tracking Work**

- Create, label, assign, and comment on **issues** to capture bugs, features, or tasks.
- Use **milestones** to group issues for a release.

## **Pull Requests (PRs) – Code Review Workflow**

- Submit a PR to propose changes; reviewers can comment, approve, or request modifications.
- **Merge strategies** (merge commit, squash, rebase) control history shape.

## **CI/CD with GitHub Actions**

- Define workflows in a .github/workflows/ YAML file.
- Trigger builds on events like push, pull\_request, or on a schedule.
- Reuse **actions** from the Marketplace or write custom ones.

## **Project Management**

- **Projects (Kanban boards)** organize issues/PRs into columns (To Do, In Progress, Done).
- **Roadmaps** visualize upcoming milestones and epics.

## **File Sharing & Collaboration**

- **Release assets** let you attach binaries, documentation, or other files to a tagged version.
- **GitHub Pages** host static sites directly from a repository.

## **Security Features**

Feature	Description
<b>Dependabot</b>	Automatically opens PRs to update vulnerable dependencies.
<b>Secret Scanning</b>	Detects exposed credentials in code and alerts repo owners.
<b>Code Scanning</b>	Runs static analysis tools and surfaces security findings in PRs.

### Branch Protection

Enforces status checks, required reviews, and restricts force-pushes.



## Upcoming Topics (Tomorrow & Beyond)

1. **Deep dive into GitHub** – detailed exploration of the platform’s UI and API.
2. **Creating users & managing organizations** – step-by-step setup and best practices.
3. **Issues & Pull Requests** – advanced workflows, templates, and automation.
4. **CI/CD pipelines** – building, testing, and deploying with GitHub Actions.
5. **Project management tools** – using boards, milestones, and roadmaps effectively.
6. **File sharing & security** – leveraging releases, Pages, and built-in security scans.
7. **Popular Git commands** – essential commands for daily work and interview prep.



## Share the Knowledge

If you know anyone interested in **DevOps** who hasn’t discovered this channel, invite them to join the community.